

目 录

第 1 章 安装和构建 JBoss 服务器	1
1.1 获得二进制文件	1
1.1.1 预备条件	1
1.1.2 安装二进制存档	2
1.1.3 default 服务器配置文件集合	3
1.2 基本安装测试	8
1.3 从网络服务器启动	10
1.4 基于源代码构建服务器	14
第 2 章 JBoss JMX 微内核 (Microkernel)	21
2.1 JMX 介绍	21
2.1.1 装配层	22
2.1.2 代理层	23
2.1.3 分布式服务层	23
2.1.4 JMX 组件介绍	23
2.2 JBoss JMX 实现架构	26
2.2.1 JBoss 类装载机架构	26
2.2.2 类装载和 Java 中的类型	26
2.2.3 JBoss XMBean	47
2.3 连接到 JMX 服务器	53
2.3.1 浏览服务器——JMX 控制台 Web 应用	53
2.3.2 使用 RMI 连接到 JMX	56
2.3.3 命令行方式访问 JMX	66
2.3.4 使用任何协议连接到 JMX	71
2.4 将 JMX 作为微内核	71
2.4.1 启动过程	71
2.4.2 JBoss MBean 服务	72
2.4.3 开发 JBoss MBean 服务	81
2.4.4 部署排序和依赖性	110
2.5 JBoss 部署器架构	127
2.6 借助于 SNMP 展示 MBean 事件	131
2.6.1 SNMP 适配器服务	131
2.6.2 陷阱服务事件	132
2.7 远程访问服务——分离式 Invoker	132
2.7.1 分离式 Invoker 实例: MBeanServer Invoker 适配器服务	137
2.7.2 分离式 Invoker 参考	143

第 3 章 JBoss 之命名——JNDI 命名服务	149
3.1 JNDI 概述	149
3.1.1 JNDI 应用编程接口	149
3.1.2 J2EE 和 JNDI——应用组件环境	151
3.2 JBossNS 架构	164
3.2.1 命名 InitialContext 工厂	166
3.2.2 基于 HTTP 访问 JNDI	170
3.2.3 保护基于 HTTP 访问 JNDI	176
3.2.4 保护只读、未保护上下文 JNDI 的访问	178
3.2.5 其他命名 MBean	179
第 4 章 JBoss 之事务——JTA 事务服务	185
4.1 事务/JTA 概述	185
4.1.1 悲观锁/乐观锁	186
4.1.2 分布式事务的组件	186
4.1.3 两阶段 XA 协议	187
4.1.4 启发式异常	187
4.1.5 事务 ID 和分支	187
4.2 JBoss 事务内核	188
4.2.1 为 JBoss 适配事务管理器	188
4.2.2 默认事务管理器	189
4.2.3 UserTransaction 支持	189
第 5 章 JBoss 之 EJB——EJB 容器配置和架构	191
5.1 EJB 客户端视图	191
5.2 EJB 服务器端视图	198
5.3 EJB 容器	204
5.3.1 EJBDeployer MBean	204
5.3.2 容器插件式框架	215
5.4 实体 Bean 锁和死锁检测	227
5.4.1 JBoss 为什么需要锁	227
5.4.2 实体 Bean 的生命周期	228
5.4.3 默认锁行为	228
5.4.4 插入式拦截器和锁策略	229
5.4.5 死锁	230
5.4.6 高级配置和调优	232
5.4.7 在群集中运行	236
5.4.8 修理故障	236
第 6 章 JBoss 之消息——JMS 配置和架构	239
6.1 JMS 实例	239
6.1.1 点对点实例	240

6.1.2	发布/订阅实例.....	242
6.1.3	使用持久 topic 的发布/订阅实例.....	249
6.1.4	使用 MDB 的点对点实例.....	252
6.2	JBoss 消息概述.....	261
6.2.1	调用层.....	261
6.2.2	安全性管理器.....	262
6.2.3	目的地管理器.....	263
6.2.4	消息缓存.....	263
6.2.5	状态管理器.....	263
6.2.6	持久化管理器.....	263
6.2.7	目的地.....	264
6.3	JBoss 消息配置和 MBean	264
6.3.1	org.jboss.mq.il.jvm.JVMServerILService	265
6.3.2	org.jboss.mq.il.rmi.RMIServerILService (已丢弃)	265
6.3.3	org.jboss.mq.il.oil.OILServerILService (已丢弃)	266
6.3.4	org.jboss.mq.il.uil.UILServerILService (已丢弃)	266
6.3.5	org.jboss.mq.il.uil2.UILServerILService	267
6.3.6	org.jboss.mq.il.http.HTTPServerILService	269
6.3.7	org.jboss.mq.server.jmx.Invoker	270
6.3.8	org.jboss.mq.server.jmx.InterceptorLoader	270
6.3.9	org.jboss.mq.sm.file.DynamicStateManager	270
6.3.10	org.jboss.mq.security.SecurityManager	271
6.3.11	org.jboss.mq.server.jmx.DestinationManager	273
6.3.12	org.jboss.mq.server.MessageCache	274
6.3.13	org.jboss.mq.pm.file.CacheStore.....	274
6.3.14	org.jboss.mq.pm.file.PersistenceManager	274
6.3.15	org.jboss.mq.pm.rollinglogged.PersistenceManager	275
6.3.16	org.jboss.mq.pm.jdbc2.PersistenceManager.....	275
6.3.17	目的地 MBean	276
6.3.18	借助于 JMX 管理.....	279
6.4	指定 MDB JMS 供应商	280
6.4.1	org.jboss.jms.jndi.JMSProviderLoader MBean.....	281
6.4.2	org.jboss.jms.asf.ServerSessionPoolLoader MBean	282
6.4.3	集成其他 JMS 供应商.....	283
第 7 章	JBoss 之连接器——JCA 配置和架构	285
7.1	JCA 概述	285
7.2	JBossCX 架构概述	287
7.2.1	BaseConnectionManager2 MBean.....	288
7.2.2	RARDeployment MBean	289
7.2.3	JBossManagedConnectionPool MBean	290

7.2.4	CachedConnectionManager MBean.....	291
7.2.5	JCA 资源适配器实例纲要.....	291
7.3	配置 JCA 适配器.....	298
7.3.1	配置 JDBC 数据源.....	299
7.3.2	配置常见 JCA 适配器.....	303
7.3.3	配置实例.....	306
第 8 章	JBoss 之安全性 ——J2EE 安全性配置和架构.....	307
8.1	J2EE 安全性声明概述.....	307
8.1.1	安全性引用.....	309
8.1.2	安全性身份.....	310
8.1.3	安全性角色.....	311
8.1.4	EJB 方法许可.....	312
8.1.5	Web 内容安全性约束.....	315
8.1.6	使用 JBoss 中的安全性声明.....	316
8.2	JAAS 介绍.....	316
8.3	JBoss 安全性模型.....	321
8.4	JBoss 安全性扩展架构.....	328
8.4.1	JaasSecurityManager 如何使用 JAAS.....	329
8.4.2	JaasSecurityManagerService MBean.....	331
8.4.3	扩展 JaasSecurityManager, JaasSecurityDomain MBean.....	333
8.4.4	基于 XML 的 JAAS 登录配置 MBean.....	334
8.4.5	JAAS 登录配置管理 MBean.....	336
8.4.6	使用 and 开发 JBossSX 登录模块.....	336
8.4.7	开发自定义登录模块.....	347
8.5	安全远程密码协议.....	357
8.5.1	为 SRP 提供密码信息.....	360
8.5.2	深入 SRP 算法.....	362
8.6	使用 Java 2 安全性管理器运行 JBoss.....	367
8.7	使用 JSSE 为 JBoss 提供 SSL.....	369
8.8	配置用于防火墙后的 JBoss.....	376
8.9	如何保护 JBoss 服务器.....	377
8.9.1	jmx-console.war.....	377
8.9.2	web-console.war.....	377
8.9.3	http-invoker.sar.....	377
8.9.4	jmx-invoker-adaptor-server.sar.....	377
第 9 章	集成 Servlet 容器.....	379
9.1	AbstractWebContainer 类.....	379
9.1.1	AbstractWebContainer 契约.....	380
9.1.2	创建 AbstractWebContainer 子类.....	386

9.2	JBoss/Tomcat-4.1.x 绑定	388
9.2.1	嵌入式 Tomcat 配置元素	389
9.2.2	JBoss/Tomcat 绑定使用 SSL	394
9.2.3	为 JBoss/Tomcat-4.x 绑定配置虚拟主机	401
9.2.4	使用外部静态内容	403
9.2.5	为 JBoss/Tomcat-4.x 绑定使用 Apache	405
9.2.6	使用群集	407
第 10 章	MBean 服务杂记	409
10.1	系统属性管理	409
10.2	属性编辑器管理	410
10.3	服务绑定管理	410
10.4	定时任务	417
10.5	JBoss 日志功能框架	420
10.6	RMI 动态类装载	421
第 11 章	CMP 引擎	423
11.1	启程	423
11.2	jbossCMP-jdbc 结构	430
11.3	实体 Bean	431
11.4	容器管理持久域	437
11.4.1	容器管理持久域抽象访问方法	437
11.4.2	容器管理持久域声明	438
11.4.3	容器管理持久域列映射	438
11.4.4	read-only 域	440
11.4.5	评审实体 Bean 访问	440
11.4.6	依赖值类	442
11.5	容器管理关系	446
11.5.1	cmr-field 抽象访问方法	446
11.5.2	关系声明	447
11.5.3	关系映射	448
11.6	查询	454
11.6.1	finder 和 ejbSelect 声明	455
11.6.2	EJB-QL 声明	455
11.6.3	覆盖 EJB-QL 到 SQL 的映射	457
11.6.4	JBossQL	457
11.7	优化装载	464
11.7.1	装载场景	465
11.7.2	装载组	466
11.7.3	read-ahead	467
11.7.4	装载过程	471

11.7.5 事务.....	477
11.8 乐观锁	480
11.9 实体命令和主键生成	485
11.10 Defaults.....	488
11.11 自定义数据源.....	491
11.11.1 函数映射	493
11.11.2 类型映射	493
11.11.3 用户类型映射	494
第 12 章 Web 服务	495
12.1 XDoclet.....	495
12.2 将 Hello World EJB 发布为 Web 服务	496
附录 A JBoss Group 和我们的 LGPL 授权	507
附录 B JBoss DTDs	518
附录 C 实例安装	519
附录 D 索引	520

第 1 章 安装和构建 JBoss 服务器

JBoss，免费的、兼容于 J2EE 标准的应用服务器，是目前市场上应用最广泛的开源产品。JBoss 灵活高效、易于使用的服务架构，使得它成为初学者在学习 J2EE 过程中首选的服务器平台。甚至，高级架构师也使用 JBoss 来定制其中间件平台。另外，JBoss 发布版还集成了 Servlet 容器，即 Tomcat 或者 Jetty。每当新版 JBoss 发布时，通过 SourceForge 网站 (<http://sourceforge.net/projects/jboss>)，开发者还可以找到相应的源代码，这使得调试 JBoss 服务器成为可能。另外，开发者还可以学习到 JBoss 的内部工作机理，并构建适合自己（或商业用途）的定制版本。

本章通过具体实例阐述如何安装和配置 JBoss 3.2.x 服务器。总而言之，开发者将学到如下几方面的知识：

- 从 JBoss SourceForge 项目网站获得持续更新的二进制版本
- 安装二进制版本
- 测试安装

还包括如下知识：

- 安装目录结构
- 管理员用于设置 JBoss 的若干主要配置文件
- 如何从 SourceForge CVS 库获得 3.2.x 发布版的源代码
- 如何构建 JBoss 服务器发布版

1.1 获得二进制文件

从 SourceForge JBoss 项目的“Files”页面可以获得最新的 JBoss 发布版，具体的网址位于：

<http://sourceforge.net/projects/jboss>

同时，开发者还将找到 JBoss 的历史、beta 版及 RC 版本。

1.1.1 预备条件

开发者在安装和运行 JBoss 服务器之前，必须保证目标系统安装了 JDK 1.3 以上（包括 JDK 1.3）版本。执行 `java -version` 命令除能验证上述需求是否满足外，还能确认 Java 命令工具是否位于 `path` 环境变量之中。比如，在提供 Sun 1.3.1 JDK 的 Linux 系统中运行上述命令，将获得如下结果：

```
/tmp 1206>java -version
java version "1.3.1_03"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1_03-b03)
```

Java HotSpot(TM) Client VM (build 1.3.1_03-b03, mixed mode)

至此，本文还未提及如何安装 JBoss。有一点请开发者注意，即当采用 Sun 公司提供的 Java 虚拟机时，如果 JBoss 安装目录路径上存在空格，则在某些情况下会出现错误。这是由于 Sun JVM 不能正确处理文件 URL 中包含的空格的缘故。另外，JBoss 服务器启动的各种服务所占用的端口都不在 0~1 023 范围内。因此，在 UNIX/Linux 操作系统下，并不具有 root 身份的用户就能够运行 JBoss。

1.1.2 安装二进制存档

在准备好二进制发布版后，使用 JDK 提供的 jar 工具（或其他任意 zip 解压工具）将 jboss-3.2.3.zip 存档内容解压到特定的位置。jboss-3.2.3.tgz 存档是兼容于 gnutar 格式，通过 gzip 方式存储的文件。其中，这种 tar 文件能够处理存档中的长目录名称。目前，Solaris 和 OS X 平台上的默认 tar 二进制存档不支持长目录名称。解压完成后，将创建 jboss-3.2.3 目录。下面将给出该目录介绍。

正如上文所述，安装 JBoss 发布版创建的 jboss-3.2.3 目录包含了服务器启动脚本、jar 文件、服务器配置集合及工作目录。为了能够完成程序编译，或更新配置、部署应用等工作，开发者必须了解发布版各 jar 文件的存放位置。图 1-1 阐述了 JBoss 服务器的安装目录结构。

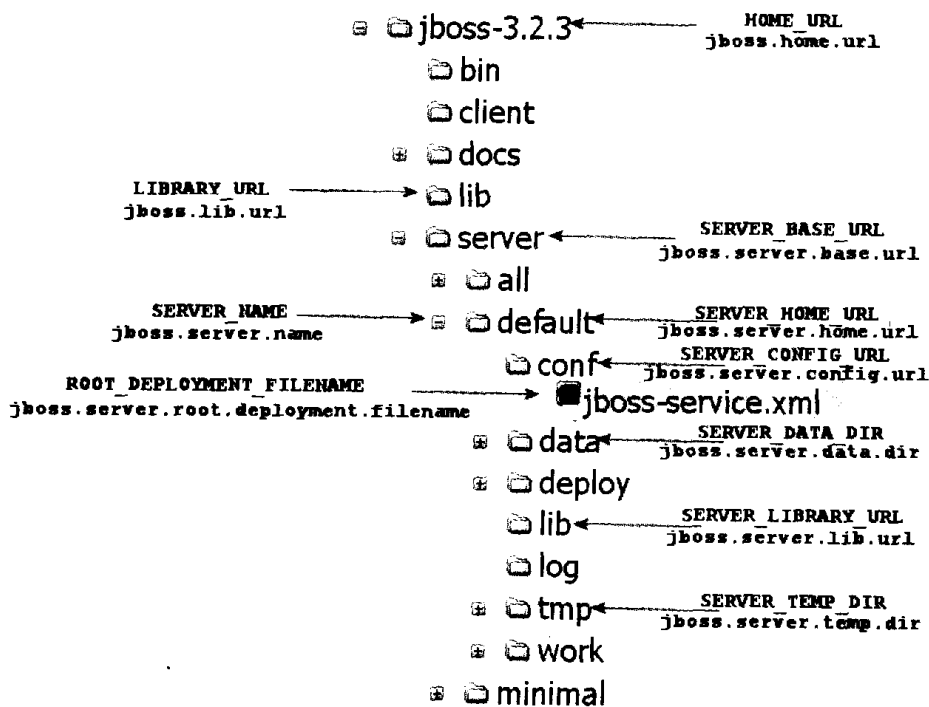


图 1-1 JBoss 的安装目录结构视图（default 服务器配置文件集合，并标明了能够被覆盖的位置信息）

本书将用 JBOSS_DIST 代指 jboss-3.2.3 目录。图 1-1 为展开的默认（default）服务器配置文件集合。其中包括子目录：conf、data、deploy、lib、log 及 tmp。初次安装后，目录中仅仅包含 conf、delpoy 及 lib 目录。表 1-1 给出了各子目录的介绍。其中，“ServerConfig 属性”是指 org.jboss.system.server.ServerConfig 中的接口常量及对应的系统属性字符串。

图 1-1 中的灰色文字描述出 ServerConfig 常量名及对应的系统属性名。XXX_URL 名给出了能够采用 URL 指定、并能访问到远程的位置信息，比如，通过 HTTP URL 能够访问到 Web 服务器。开发者能够使用表 1-1 给出的属性列表覆盖 JBoss 发布版的默认配置。

表 1-1 各子目录介绍

目 录	描 述	ServerConfig 属性
bin	启动 JBoss 所需的所有入口级 jar 文件及在 JBoss 发布版中附带的启动脚本都位于 bin 目录下	无
client	客户端应用所需的 jar 文件都存放在 client 目录。在通常情况下，客户程序需要如下几个 jar 文件： <ul style="list-style-type: none"> ● jbossall-client.jar ● concurrent.jar ● log4j.jar ● jaas.jar, jnet.jar (如果没有使用 JDK 1.4 以上版本) ● jcert.jar, jsse.jar (为 SSL 服务，如果没有使用 JDK 1.4 以上版本) 	无
server	JBoss 服务器配置集合位于 server 服务下。默认的服务器配置集合为 server/default。JBoss 带有 minimal、default 及 all 配置集合。“1.1.3 default 服务器配置文件集合”中的图 1-2 将给出 minimal 和 default 配置集合各子目录及重要配置文件的讨论	SERVER_BASE_DIR= "jboss.server.base.dir" SERVER_BASE_URL= "jboss.server.base.url"
lib	lib 目录包含了 JBoss 使用的启动 jar 文件。其中，开发者不要在该目录下放置自己的任何库文件	LIBRARY_URL= "jboss.lib.url"
conf	conf 目录存放了 bootstrap 配置描述符（在默认情况下为 jboss-server.xml 文件），用于特定的服务器配置。它定义了服务器生命周期中固有的核心服务	SERVER_CONFIG_URL= "jboss.server.config.url"
data	data 目录为需要在文件系统中存储内容的服务提供方便	SERVER_DATA_DIR= "jboss.server.data.dir"
deploy	deploy 目录是 JBoss 用于寻找动态部署内容（即热部署服务）的默认位置。通过覆盖 URLDeploymentScanner 的 URLs 属性可以达到修改默认位置的目的	无
lib	lib 目录是 bootstrap 部署描述符制定的默认位置。该目录下的所有 jar 文件将被装载到共享的 classpath 中	SERVER_LIBRARY_URL= "jboss.server.lib.url"
log	bootstrap 日志服务在默认情况下将 log 目录作为日志的存放地。通过修改 conf/log4j.xml 配置文件能够修改默认存放地	无
tmp	将部署的应用拷贝到 tmp 目录后，能够供本地使用	SERVER_TEMP_DIR= "jboss.server.temp.dir"

1.1.3 default 服务器配置文件集合

JBoss_DIST/server 目录包含一个或多个配置文件集合。default 服务器配置文件集合位于目录 JBoss_DIST/server/default 下。JBoss 允许开发者添加多个配置集合，从而可以

很容易地更换不同配置来运行服务器。在通常情况下，为创建新的配置文件集合，开发者只需要将 default 服务器配置文件集合复制到新的目录中，然后修改目标配置文件即可。图 1-2 描述了 default 服务器配置文件集合所包含的内容。

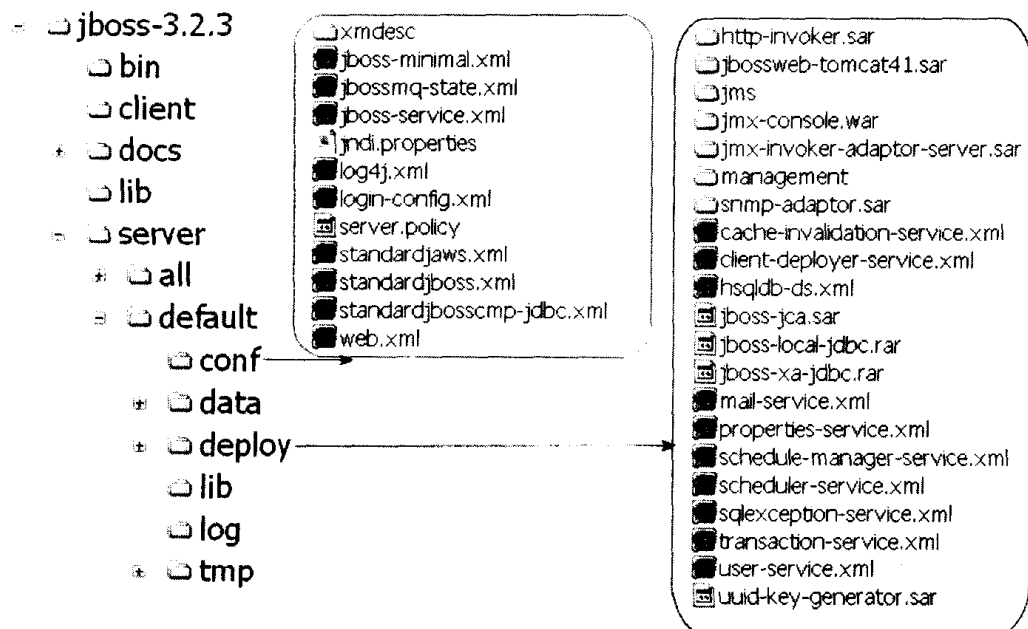


图 1-2 default 服务器配置文件集合中 conf 和 deploy 目录展开视图

1. conf/jboss-minimal.xml

这是 jboss-service.xml 配置文件所允许的最小实例。它对应于 minimal 配置文件集合中的 jboss-service.xml 文件。

2. conf/jboss-service.xml

jboss-service.xml 定义了核心服务配置。“2.4.2 JBoss MBean 服务”节的内容描述了该配置文件的完整 DTD 和语法，以及集成定制服务的具体细则。

3. conf/jbossmq-state.xml

jbossmq-state.xml 为 JBossMQ 配置文件。它除了用于描述用户名和密码的映射关系外，还给出了那些持久性订阅的用户。其中，jbossmq-state.xml 文件格式可以通过“6.3.9 org.jboss.mq.sm.file.DynamicStateManager”节的内容找到。

4. conf/jndi.properties

在 JBoss 服务器中，当没有提供参数创建 InitialContext 时，JNDI InitialContext 属性需要使用 jndi.properties 文件。

5. conf/log4j.xml

由于 JBoss 服务器代码使用了 Apache Log4j 框架，因此为配置 category 优先级和 appender，开发者可以通过 log4j.xml 实现。在 JBoss 中如何配置和使用 Log4j，请参考《Using Log4j with JBoss》一书。

6. conf/login-config.xml

为实现服务器端基于 JAAS 的安全性认证配置, login-config.xml 文件提供了实例实现。其中,“第 8 章 JBoss 之安全性——J2EE 安全性配置和架构”详细阐述了 JBoss 安全性框架的其他细则和该文件的格式。

7. conf/server.policy

server.policy 文件存放了 Java 2 安全性许可。JBoss 提供的默认文件简单地将所有的许可授给了所有的 codebase¹。

8. conf/standardjaws.xml

standardjaws.xml 文件为遗留 EJB 1.1 JBossCMP 引擎提供了默认配置入口。在 JBoss 3.0 中为支持 EJB 2.0, 重新开发了容器管理持久化 (Container-Managed Persistence, CMP) 层。另外, 针对 JBoss 3.2 版, 本书还重写了“CMP 引擎”(见第 11 章)。

9. conf/standardjbosscmp-jdbc.xml

standardjbosscmp-jdbc.xml 文件为 JBoss 3.2 中 EJB 2.0 JBossCMP 引擎提供了默认配置入口。其具体细节, 请参考“第 11 章 CMP 引擎”。

10. conf/standardjboss.xml

standardjboss.xml 文件提供了默认的容器配置。其中, 该文件的具体使用请参考“第 5 章 JBoss 之 EJB——EJB 容器配置和架构”。

11. deploy/cache-invalidation-service.xml

cache-invalidation-service.xml 是这样一种服务, 即它为借助于 JMS 通知而实现 EJB 缓存提供了自定义有效性的配置入口。在默认情况下, 它处于失效状态。

12. deploy/client-deployer-service.xml

client-deployer-service.xml 是这样一种服务, 即它为 J2EE 应用客户提供支持。它基于客户应用提供的 application-client.xml 描述符, 而负责管理“java:comp/env”企业命名上下文。

13. deploy/hsqldb-ds.xml

hsqldb-ds.xml 是嵌入式 Hypersonic 1.7.1 数据库的配置文件。它设置该嵌入式数据库及相关连接工厂。“7.3.1 配置 JDBC 数据源”节内容有 JCA 数据源文件格式的深入研究。

14. deploy/http-invoker.sar

http-invoker.sar 包含支持 RMI/HTTP 方式传输的分离式 Invoker。与此同时, 它为基于 HTTP 方式访问 JBoss JNDI 服务的 JNDI 命名服务设置了 RMI/HTTP 代理绑定。在 2.7.2 中的“HttpInvoker——RMI/HTTP 传输”小节将对此进行深入讨论。

¹译者注: 在使用 Java 语言编写的程序中, 不仅可以在本地 classpath 中加载类, 还可以根据需从网络上下载类。为了使 Java 程序可以从网络上下载类, 开发者需要使用 codebase。codebase 指定了 Java 程序在网络上何处可以获得所需要的类。

15. deploy/jboss-jca.sar

jboss-jca.sar 是 JCA 1.0 规范的应用服务器实现。它为 JBoss 服务器集成资源适配器提供了连接管理服务。“第 7 章 JBoss 之连接器——JCA 配置和架构”有更深入的讨论。

16. deploy/jboss-local-jdbc.rar

jboss-local-jdbc.rar 是 JCA 资源适配器。它为支持 DataSource 接口、但不提供 JCA 的 JDBC 驱动实现了 JCA ManagedConnectionFactory 接口。

17. deploy/jboss-xa.rar

jboss-xa.rar 也是一种 JCA 资源适配器。它为支持 XADataSource 接口、但不提供 JCA 适配器的 JDBC 驱动实现了 JCA ManagedConnectionFactory 接口。

18. deploy/jbossweb-tomcat41.sar

jbossweb-tomcat41.sar 目录为配置 Tomcat 4.1.27 的 Servlet 引擎提供了未打包的 MBean 服务存档。由于 SAR 是以文件夹的形式展开的，而没有以 jar 存档的方式部署，使得修改 jbossweb-tomcat41.sar/META-INF/jboss-service.xml 部署描述符很方便。进一步的讨论，请参考“第 9 章 集成 Servlet 引擎”的内容。

19. deploy/jms/jbossmq-destinations-service.xml

jbossmq-destinations-service.xml 文件为 JMS 单元测试配置了若干个 JMS 队列（Queue）和主题（Topic）。“第 6 章 JBoss 之消息——JMS 配置和架构”中的内容有 JMS 目的地（destination）配置的细则。

20. deploy/jms/jbossmq-httpil.sar

jbossmq-httpil.sar 为实现 HTTP 方式使用 JMS 而提供了 JMS 调用层。

21. deploy/jms/jbossmq-service.xml

jbossmq-service.xml 文件配置核心 JBossMQ JMS 服务。“第 6 章 JBoss 之消息——JMS 配置和架构”中有 JMS 服务的详细讨论。

22. deploy/jms/jms-ra.rar

jms-ra.rar 是一种 JCA 资源适配器。它为 JMS 连接工厂实现了 JCA ManagedConnectionFactory 接口。

23. deploy/jms/jms-ds.xml

jms-ds.xml 文件为同 jms-ra.rar JCA 资源适配器一起使用而提供了 JBossMQ JMS 供应商配置入口。

24. deploy/jms/jvm-il-service.xml

jvm-il-service.xml 为同一 JVM 中 JMS 传输调用层提供了配置入口。“6.3.1 org.jboss.mq.il.jvm.JVMServerILService”节内容有关于传输层的深入讨论。

25. deploy/jms/oil-service.xml

oil-service.xml 为 JMS 优化调用层提供了配置入口。“6.3.3 org.jboss.mq.il.oil.OILServerIL

Service (已丢弃)” 有关于传输层的进一步讨论。

26. deploy/jms/oil2-service.xml

oil2-service.xml 为第二版 JMS 优化调用层提供了配置入口。不要将它应用于实际, JBoss 将丢弃它。

27. deploy/jms/rmi-il-service.xml

rmi-il-service.xml 为基于 JMS RMI 调用层提供了配置入口。由于它的传输速度很慢, 因此建议开发者不要使用, 而且 JBoss 将会丢弃它。

28. deploy/jms/uil2-service.xml

uil2-service.xml 为配置第二版 JMS 统一调用层提供了配置入口。这是一种基于套接字的自定义传输方式, 速度最快, 并且最为可靠, 因此本书建议将它用于消息调用。“6.3.5 org.jboss.mq.il.uil2.UILServerILService” 节内容有关于传输层的详细讨论。

29. deploy/jmx-console.war

jmx-console.war 目录是一个展开的 Web 应用存档, 它为 JMX MBeanServer 提供了 HTML 适配器。考虑到将来通过 jmx-console.war/WEB-INF/*.xml 部署描述符配置基于角色的安全性的便利, 该 WAR 并不通过压缩存档部署, 而是以展开的形式存在。jmx-console 的进一步讨论, 请参考“2.3.1 浏览服务器——JMX 控制台 Web 应用”节的内容。

30. deploy/jmx-invoker-adaptor-server.sar

jmx-invoker-adaptor-server.sar 是以展开方式存在的 MBean 服务存档, 它将 JMX MBeanServer 接口的部分方法作为 RMI 接口, 使得 JMX 核心功能能够被远程访问。它和遗留 jmx-rmi-adaptor.sar 在这方面很类似, 不同之处在于 jmx-invoker-adaptor-server 是通过分离式 Invoker 架构处理传输的。“2.3.4 使用任何协议连接到 JMX” 节有深入的讨论。

31. deploy/mail-service.xml

mail-service.xml 是 MBean 服务描述符, 它为在 JBoss 服务器内部使用 JavaMail 会话服务提供了配置入口。

32. deploy/management/console-mgr.sar, web-console.war

console-mgr.sar 和 web-console.war 为实验性的 Web 应用/Applet, 它们比 jmx-console.war 提供了更丰富的 JMX 服务器管理数据视图。目前, 它们还处于开发阶段。通过使用 URL <http://localhost:8080/web-console/> 可以查看该控制台。

33. deploy/properties-service.xml

properties-service.xml 是 MBean 服务配置描述符。它既能够定义系统属性, 还能够自定义 JavaBean 属性编辑器 (PropertyEditor)。“10.1 系统属性管理” 节内容有它的进一步研究和讨论。

34. deploy/scheduler-service.xml, schedule-manager-service.xml

scheduler-service.xml 和 schedule-manager-service.xml 文件是 MBean 服务配置描述符。它们提供了一种定时服务类型。更进一步的研究, 请参考“10.4 定时任务” 节内容。

35. deploy/snmp-adaptor.sar

snmp-adaptor.sar 为 JMX 提供 SNMP 适配器。它能够将 JMX 通知映射到 SNMP Trap。

36. deploy/sqlexception-service.xml

sqlexception-service.xml 文件为 MBean 服务配置描述符，它用于处理与特定数据库厂商相关的 `java.sql.SQLException` 异常。具体的使用方法，请参考“11.9 实体命令和主键生成”节内容的进一步研究。

37. deploy/transaction-service.xml

transaction-service.xml 配置描述符用于设置 JBoss JTA 事务管理器相关服务。更进一步的讨论，请参考“第4章 JBoss 之事务——JTA 事务服务”的内容。

38. deploy/user-service.xml

user-service.xml 文件为开发者自定义 MBean 服务，它提供了 MBean 服务配置描述符的模板。然而，开发者通常都不会使用到该文件。“2.4.3 开发 JBoss MBean 服务”的内容有开发 MBean 服务的详细讨论。

39. deploy/uuid-key-generator.sar

uuid-key-generator.sar 服务提供了一种基于 UUID 的键生成策略。

1.2 基本安装测试

在完成 JBoss 发布版的安装后，开发者最好能够对它进行简单的启动测试。这样就可以知道，运行在特定 JVM 和操作系统组合上的 JBoss 有没有严重的错误。为实现测试的目的，开发者需要将当前目录转到 `JBoss_DIST/bin`，然后针对不同操作系统执行相应的 `run.bat` 或 `run.sh` 脚本。如下给出了输出结果实例，以确保不存在异常消息。

```
[nr@toki bin]$ ./run.sh
=====

JBoss Bootstrap Environment

JBoss_HOME: /tmp/jboss-3.2.3

JAVA: /System/Library/Frameworks/JavaVM.framework/Home/bin/java

JAVA_OPTS: -Dprogram.name=run.sh

CLASSPATH: /tmp/jboss-3.2.3/bin/run.jar:/System/Library/Frameworks/
JavaVM.framework/Home/
/lib/tools.jar
=====

11:53:32,324 INFO [Server] Starting JBoss (MX MicroKernel)...
```



```
11:53:32,407 INFO [Server] Release ID: JBoss [WonderLand] 3.2.3 (build:
CVSTag=JBoss_3_2_3 date=200311301445)
11:53:32,415 INFO [Server] Home Dir: /private/tmp/jboss-3.2.3
11:53:32,417 INFO [Server] Home URL: file:/private/tmp/jboss-3.2.3/
11:53:32,417 INFO [Server] Library URL: file:/private/tmp/jboss-3.2.3/lib/
11:53:32,421 INFO [Server] Patch URL: null
11:53:32,424 INFO [Server] Server Name: default
11:53:32,458 INFO [Server] Server Home Dir: /private/tmp/jboss-3.2.3/server/default
11:53:32,499 INFO [Server] Server Home URL: file:/private/tmp/jboss-3.2.3/server/default/
11:53:32,500 INFO [Server] Server Data Dir: /private/tmp/jboss-3.2.3/server/default/data
11:53:32,501 INFO [Server] Server Temp Dir: /private/tmp/jboss-3.2.3/server/default/tmp
11:53:32,501 INFO [Server] Server Config URL: file:/private/tmp/jboss-3.2.3/server/default/conf/
11:53:32,503 INFO [Server] Server Library URL: file:/private/tmp/jboss-3.2.3/server/default/lib/
11:53:32,504 INFO [Server] Root Deployment Filename: jboss-service.xml
11:53:32,518 INFO [Server] Starting General Purpose Architecture (GPA)...
11:53:33,511 INFO [ServerInfo] Java version: 1.4.2_03, Apple Computer, Inc.
11:53:33,512 INFO [ServerInfo] Java VM: Java HotSpot(TM) Client VM 1.4.2-34, "Apple Computer,
Inc."
11:53:33,512 INFO [ServerInfo] OS-System: Mac OS X 10.3.3, ppc
11:53:33,648 INFO [ServiceController] Controller Mbean online
11:53:33,963 INFO [MainDeployer] Started jboss.system:service=MainDeployer
11:53:34,263 INFO [MainDeployer] Adding deployer: org.jboss.deployment.JARDeployer@1a700a
11:53:34,264 INFO [JARDeployer] Started jboss.system:service=JARDeployer
11:53:34,328 INFO [MainDeployer] Adding deployer: org.jboss.deployment.SARDeployer@b68d78
11:53:34,386 INFO [SARDeployer] Started jboss.system:service=ServiceDeployer
11:53:34,387 INFO [Server] Core system initialized
```

如果输出信息与此类似（排除安装目录的差异性），则开发者可以开始使用 JBoss。为关闭 JBoss 服务器，只需在运行 JBoss 的控制台中敲入【Ctrl】+【C】组合键。或者，可以使用 shutdown.sh 命令行工具：

```
[nr@toki bin]$ ./shutdown.sh
A JMX client to shutdown (exit or halt) a remote JBoss server.

Usage: shutdown [options] <operation>

options:
-h, --help          Show this help message
-D<name>[=<value>]  Set a system property
--                  Stop processing options
-s, --server=<url>   Specify the JNDI URL of the remote server
-n, --serverName=<url> Specify the JMX name of the ServerImpl
-a, --adapter=<name> Specify JNDI name of the RMI adapter to use
-u, --user=<name>    Specify the username for authentication[not implemented yet]
-p, --password=<name> Specify the password for authentication[not implemented yet]

operations:
```

```
-S, --shutdown      Shutdown the server (default)
-e, --exit=<code>    Force the VM to exit with a status code
-H, --halt=<code>    Force the VM to halt with a status code
```

如果在运行 JBoss 的过程中并没有为 `run.sh` 提供任何参数，则服务器使用 default 服务器配置文件集合。为通过不同的配置文件集合启动服务器，开发者需要将 `JBOSS_DIST/server` 目录下的配置文件名传递给 `-c` 命令行选项。比如，以 `minimal` 配置文件集合启动 JBoss，过程如下：

```
[nr@toki bin]$ ./run.sh -c minimal
...
12:00:26,983 INFO [Server] JBoss (MXMicroKernel) [3.2.3 (build: CVSTag=JBoss_3_2_3 date=200311301445)]
Started in 3s:343ms
```

为了查看 `run.bat`（或 `run.sh`）运行 JBoss 服务器所支持的所有命令行选项，开发者可以通过 `-h` 选项获得详细介绍如下：

```
usage: run.bat [options]

options:
  -h, --help                Show this help message
  -V, --version              Show version information
  --                        Stop processing options
  -D<name>[=<value>]        Set a system property
  -p, --patchdir=<dir>      Set the patch directory; Must be absolute
  -n, --netboot=<url>       Boot from net with the given url as base
  -c, --configuration=<name> Set the server configuration name
  -j, --jaxp=<type>         Set the JAXP impl type (ie. Crimson)
  -L, --library=<filename>  Add an extra library to the loaders classpath
  -C, --classpath=<url>     Add an extra url to the loaders classpath
  -P, --properties=<url>    Load system properties from the given url
  -b, --host=<host or ip>   Bind address for all JBoss services
```

1.3 从网络服务器启动

“`--netboot=url`”是一种非常实用的命令行选项。它能够触发 JBoss 使用给定的 `url` 作为基本 `url`，继而从此处装载所有的类库和配置。通过网络启动选项能够将 `Server Config.HOME_URL` 设置为“`--netboot`”选项传递的 `url` 参数值。如果命令行没有指定任何其他选项，标准 `JBOSS_DIST` 结构中的所有其他位置能够根据 `HOME_URL` 值依次类推。其具体含义为，如果 Web 服务器上存放了 JBoss 发布版，从本地 `JBOSS_DIST/bin` 目录启动 JBoss 服务器只会使用到本地的 `run.bat`（或 `run.sh`）脚本和 `run.jar` 文件。有一点请开发者注意，即这种 Web 服务器必须支持 `PROPFIND` WebDAV 命令。另外，JBoss 本身包括了一个简单的 `Servlet` 过滤器，它对 `PROPFIND` 命令提供了最低限度的支持，因此 JBoss 本身也能够作为网络启动 Web 服务器。

为实现从网络启动 JBoss，本书提供了一个 Ant 编译脚本用于创建定制网络启动配置文件集合，即 `examples/src/main/org/jboss/chap1/build-netboot.xml` 文件。为测试网络启动特性，需要使用 Ant 命令行工具运行 `build-netboot.xml` 脚本时，指定 `JBoss_DIST` 的位置。具体过程如下：

```
[nr@toki examples]$ ant -Djboss.dist=/tmp/jboss-3.2.3 -buildfile src/main/org/jboss/chap1/build-netboot.xml
Buildfile: src/main/org/jboss/chap1/build-netboot.xml

netboot:
[mkdir] Created dir: /tmp/jboss-3.2.3/server/netboot
[copy] Copying 32 files to /tmp/jboss-3.2.3/server/netboot
[unzip] Expanding: /tmp/jboss-3.2.3/docs/examples/netboot/netboot.war into /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war
[copy] Copying 14 files to /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war
[copy] Copying 148 files to /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default
[copy] Copying 1 file to /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/jmx-console.war/WEB-INF/lib

zipdir:
[move] Moving 10 files to /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/http-invoker.sarx
[zip] Building zip: /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/http-invoker.sar [delete] Deleting directory /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/http-invoker.sarx

zipdir:
[move] Moving 8 files to /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/jms/jbossmq-httpil.sarx
[zip] Building zip: /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/jms/jbossmq-httpil.sar
[delete] Deleting directory /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/jms/jbossmq-httpil.sarx

zipdir:
[move] Moving 24 files to /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/jbossweb-tomcat41.sarx
[zip] Building zip: /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/jbossweb-tomcat41.sar
[delete] Deleting directory /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/jbossweb-tomcat41.sarx

zipdir:
[move] Moving 23 files to /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/jmx-console.warx
[zip] Building zip: /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/
```

```
jmx-console.war [delete] Deleting directory /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/
server/default/deploy/jmx-console.warx

zipdir:
[move] Moving 2 files to /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/
jmx-invoker-adaptor-server.sarx
[zip] Building zip: /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/
jmx-invoker-adaptor-server.sar
[delete] Deleting directory /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/
jmx-invoker-adaptor-server.sarx

zipdir:
[move] Moving 6 files to /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/
snmp-adaptor.sarx
[zip] Building zip: /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/
snmp-adaptor.sar
[delete] Deleting directory /tmp/jboss-3.2.3/server/netboot/deploy/netboot.war/server/default/deploy/
snmp-adaptor.sarx

BUILD SUCCESSFUL
Total time: 10 seconds
```

然后，通过指定 **netboot** 配置文件集合启动 JBoss 服务器。具体过程如下：

```
[nr@toki bin]$ ./run.sh -c netboot
=====

JBoss Bootstrap Environment

JBOSS_HOME: /tmp/jboss-3.2.3

JAVA: /System/Library/Frameworks/JavaVM.framework/Home/bin/java

JAVA_OPTS: -Dprogram.name=run.sh

CLASSPATH: /tmp/jboss-3.2.3/bin/run.jar:/System/Library/Frameworks/JavaVM.framework/Home/
lib/tools.jar
=====

11:16:49,563 INFO [Server] Starting JBoss (MX MicroKernel)...
11:16:49,569 INFO [Server] Release ID: JBoss [WonderLand] 3.2.3 (build: CVSTag=JBoss_3_2_3
date=200311301445)
11:16:49,629 INFO [Server] Home Dir: /private/tmp/jboss-3.2.3
11:16:49,632 INFO [Server] Home URL: file:/private/tmp/jboss-3.2.3/
11:16:49,635 INFO [Server] Library URL: file:/private/tmp/jboss-3.2.3/lib/
11:16:49,642 INFO [Server] Patch URL: null
11:16:49,647 INFO [Server] Server Name: netboot
```

```
11:16:49,670 INFO [Server] Server Home Dir: /private/tmp/jboss-3.2.3/server/netboot
11:16:49,673 INFO [Server] Server Home URL: file:/private/tmp/jboss-3.2.3/server/netboot/
11:16:49,723 INFO [Server] Server Data Dir: /private/tmp/jboss-3.2.3/server/netboot/data
11:16:49,728 INFO [Server] Server Temp Dir: /private/tmp/jboss-3.2.3/server/netboot/tmp
11:16:49,731 INFO [Server] Server Config URL: file:/private/tmp/jboss-3.2.3/server/netboot/conf/
11:16:49,734 INFO [Server] Server Library URL: file:/private/tmp/jboss-3.2.3/server/netboot/lib/
11:16:49,737 INFO [Server] Root Deployment Filename: jboss-service.xml
11:16:49,749 INFO [Server] Starting General Purpose Architecture (GPA)...
11:16:50,519 INFO [ServerInfo] Java version: 1.4.2_03,Apple Computer, Inc.
11:16:50,526 INFO [ServerInfo] Java VM: Java HotSpot(TM) Client VM 1.4.2-34,"Apple Computer,
Inc."
11:16:50,529 INFO [ServerInfo] OS-System: Mac OS X 10.3.3,ppc...
11:17:05,413 INFO [MainDeployer] Deployed package: file:/private/tmp/jboss-3.2.3/server/netboot/
conf/jboss-service.xml
11:17:05,533 INFO [Server] JBoss (MX MicroKernel) [3.2.3 (build: CVSTag=JBoss_3_2_3
date=200311301445)] Started in 15s:783ms
```

最后，从 JBOSS_DIST/bin 目录启动另外一个 JBoss 实例。比如：

```
=====
```

```
JBoss Bootstrap Environment
```

```
JBOSS_HOME: /tmp/jboss-3.2.3
```

```
JAVA: java
```

```
JAVA_OPTS: -Dprogram.name=run.sh
```

```
CLASSPATH: /tmp/jboss-3.2.3/bin/run.jar:/lib/tools.jar
```

```
=====
```

```
11:15:16,810 INFO [Server] Starting JBoss (MX MicroKernel)...
11:15:16,814 INFO [Server] Release ID: JBoss [WonderLand] 3.2.3 (build: CVSTag=JBoss_3_2_3
date=200311301445)
11:15:16,815 INFO [Server] Home Dir: /private/tmp/jboss-3.2.3
11:15:16,816 INFO [Server] Home URL: http://toki.local.:8080/netboot/
11:15:16,817 INFO [Server] Library URL: http://toki.local.:8080/netboot/lib/
11:15:16,821 INFO [Server] Patch URL: null
11:15:16,822 INFO [Server] Server Name: default
11:15:16,823 INFO [Server] Server Home Dir: /private/tmp/jboss-3.2.3/server/default
11:15:16,824 INFO [Server] Server Home URL: http://toki.local.:8080/netboot/server/default/
11:15:16,825 INFO [Server] Server Data Dir: /private/tmp/jboss-3.2.3/server/default/data
11:15:16,825 INFO [Server] Server Temp Dir: /private/tmp/jboss-3.2.3/server/default/tmp
11:15:16,826 INFO [Server] Server Config URL: http://toki.local.:8080/netboot/server/default/conf/
11:15:16,827 INFO [Server] Server Library URL: http://toki.local.:8080/netboot/server/default/lib/
11:15:16,828 INFO [Server] Root Deployment Filename: jboss-service.xml
```



```
11:15:16,839 INFO [Server] Starting General Purpose Architecture (GPA)...
11:15:17,676 INFO [ServerInfo] Java version: 1.4.2_03,Apple Computer, Inc.
11:15:17,679 INFO [ServerInfo] Java VM: Java HotSpot(TM) Client VM 1.4.2-34,"Apple Computer, Inc."
11:15:17,679 INFO [ServerInfo] OS-System: Mac OS X 10.3.3,ppc...
11:16:53,266 INFO [MainDeployer] Deployed package: http://toki.local.:8080/netboot/server/default/conf/jboss-service.xml
11:16:53,272 INFO [Server] JBoss (MX MicroKernel) [3.2.3 (build: CVSTag=JBoss_3_2_3 date=200311301445)] Started in 1m:36s:432ms
```



这里定制的 netboot 配置文件集合仅仅包含运行 jbossweb-tomcat41.sar Web 服务器所需的文件及 netboot.war 文件。其中，netboot.war 包含 JBOSS_DIST/lib 和 JBOSS_DIST/server/default 目录下的所有文件。

1.4 基于源代码构建服务器

各个 JBoss 模块的源代码都能够找到。其中，从 SourceForge 下载合适版本的 JBoss 源代码后，开发者就能够构建相应版本的 JBoss 了。

1. 访问位于 SourceForge 的 JBoss CVS 存储库

JBoss 源代码存放在 SourceForge，一个由 VA Linux 系统提供的、为开源社区服务的网站。sourceforge.net 以将近 80 000 个开源项目和超过 850 000 个注册用户，而成为最大的开源服务集散地。大量的开源项目已经将各自的开发转移到 SourceForge.net 网站。SourceForge 提供的服务包括项目 CVS 存储和用于项目管理的 Web 界面。其中，Web 界面包括 bug 跟踪、发布管理、邮件列表，等等。最为关键的地方在于，所有的这些服务对于所有的开源开发者都是免费的。进一步的内容和项目情况，请参考 SourceForge 的主页 (<http://sourceforge.net>)。

2. 理解 CVS

并行版本系统 (Concurrent Versions System, CVS) 是开源的、普遍用于开源社区的工具。CVS 是源代码控制或修改控制工具，用于跟踪团队中不同开发者对相同文件所做的修改。CVS 使得团队中的开发者在独立工作的同时，还能够同步各自的工作。

3. 匿名访问 CVS

根据下文给出的操作指南，通过匿名 (以 pserver 方式) 能够访问 JBoss 项目的 SourceForge CVS 存储库。其中，必须为待 check out 的模块指定模块名。当提示输入 anonymous 用户的密码时，只需简单地按下回车键。用于匿名访问 JBoss 存储库的通用 CVS 命令行语法如下：

```
cvs -d:pserver:anonymous@cvs.jboss.sourceforge.net:/cvsroot/jboss login
cvs -z3 -d:pserver:anonymous@cvs.jboss.sourceforge.net:/cvsroot/jboss co modulename
```

第一条命令为匿名用户登录到 JBoss CVS 存储库的情形。对于特定的机器而言，该命

令只需完成一次，因为登录信息被系统保存在 HOME/.cvspass 文件中。第二条命令将指定的模块代码 check out 到运行 cvs 命令所在的目录里面。为避免每次都要敲入很长的命令行，开发者可以设置 CVSROOT 环境变量并将其值取为“:pserver:anonymous@cvs.jboss.sourceforge.net:/cvsroot/jboss”。然后，通过如下的命令行可以达到上述目的：

```
cvs login
cvs -z3 co modulename
```

开发者 check out 的 JBoss 模块别名依赖于目标 JBoss 版本。对于 3.0 分支而言，模块名为 jboss-3.0；对于 3.2 分支，模块名为 jboss-3.2。一般情况下，对于分支 x.y，模块名为 jboss-x.y。为了能够 check out JBoss 的 HEAD 修订以获得主分支的最新代码集合，开发者可以使用 jboss-head 作为模块名。JBoss 每次发布新版本时，都是通过 JBoss_X_Y_Z 模式标记的。其中，X 为主版本，Y 为次版本，Z 为补丁版本。另外，JBoss 每次发布新分支时，都是以 Branch_X_Y 模式标记的。如下给出一些实例：

```
cvs co -r Branch_3_0 jboss-3.0 # Checkout the current 3.0 branch code
cvs co -r JBoss_3_0_6 jboss-3.0 # Checkout the 3.0.6 release version code
cvs co -r Branch_3_2 jboss-3.2 # Checkout the current 3.2 branch code
cvs co -r JBoss_3_2_0 jboss-3.2 # Checkout the 3.2.0 release version code
cvs co jboss-head # Checkout the current HEAD branch code
```

4. 获得 CVS 客户端

开发者几乎可以获得任何平台的免费 CVS 命令行工具。其中，大部分 Linux 和 UNIX 发布版本默认情况下都包含了 CVS 命令行工具。对于 Win32 平台而言，在 <http://sources.redhat.com/cygwin/> 网址提供了 Cygwin 的介绍。同时，该网址还给出了几个其他 Win32 平台版的 UNIX 程序。由于 CVS 命令行版本的语法在各平台下都是相同的，通过它可以检验结果是否一致。

如果需要完整的 CVS 文档，请参考 CVS 的主页：<http://www.cvshome.org/>。

5. 使用源代码构建 JBoss-3.2.3 发布版

每个 JBoss 发布版都包含了源代码存档，其中含有构建该发布版所需要的一切。同时，通过 JBoss 项目网站 <http://sourceforge.net/projects/jboss/> 的“Files”节内容可以获得源代码。由于源代码目录结构和如下描述的 CVS 源代码树一致，所以通过下面给出的操作指南能够构建发布版。接下来，本文从获得 jboss-3.2 源代码树开始阐述。

6. 使用 CVS 源代码构建 JBoss-3.2.3 发布版

本节内容指导开发者如何从 CVS 源代码构建 JBoss 发布版。首先，创建一个目录以存放下载的 CVS 源代码，并转移到新创建的目录中。用 CVS_WD 代表该目录，以指定 CVS 工作目录。在本例子中，将源代码 check out 到 Linux 系统的 /tmp/3.2.3 目录中。其次，通过如下方式获得 3.2.3 版本的 JBoss 源代码²：

²在 JBoss 源代码发布到 3.0.4 时，为获得完成的源代码，JBoss 项目对项目别名做了修改。即，不再是为每个分支使用 jboss-all 作为模块别名，而需要指定具体的模块别名。比如，对于 3.0 分支，模块别名为 jboss-3.0；对于 3.2 分支，模块别名为 jboss-3.2，等等。如果需要 check out JBoss 的 HEAD 修订以获得主

```
[nr@toki tmp]$ mkdir 3.2.3
[nr@toki tmp]$ cd 3.2.3
[nr@toki 3.2.3]$ export CVSROOT=:pserver:anonymous@cvs.sourceforge.net:/cvsroot/jboss
[nr@toki 3.2.3]$ cvs co -r JBoss_3_2_3 jboss-3.2
cvs server: Updating tools
U tools/.classpath
U tools/.donotremove
U tools/.project
cvs server: Updating tools/apache
...
```

其结果将是，jboss-3.2 目录结构中包含了构建服务器所需的所有 CVS 模块。为完成构建，将当前目录定位到 jboss-3.2/build。然后，针对开发者操作系统的不同，执行 build.sh 或 build.bat 文件。另外，开发者还需要设置 JAVA_HOME 环境变量，以定位待用的 JDK。如列表 1-1 所示。在实例中，将 JAVA_HOME 设置为 Sun JDK 的安装目录“/home/starksm/Java/j2sdk1.4.1_02”。

列表 1-1 JBoss 3.2.x 分支构建过程

```
[orb@toki 3.2.3]$ cd jboss-3.2/build/
[orb@toki build]$ export JAVA_HOME=/System/Library/Frameworks/JavaVM.framework/Home/
[orb@toki build]$ PATH=$JAVA_HOME/bin:$PATH
[orb@toki build]$ ./build.sh
Searching for build.xml ...
Buildfile: /tmp/3.2.3/jboss-3.2/build/build.xml

...

BUILD SUCCESSFUL
Total time: 2 minutes 41 seconds
```

如果在构建过程中，出现如下的错误：

```
BUILD FAILED
file:/tmp/3.2.3/jboss-3.2/server/build.xml:233: Failed to launch JJTree
```

则表明 JAVA_HOME/bin 目录没有包含在 path 环境变量中。其中，错误的根源在于 JavaCC JJTree Ant 任务需要它。

构建过程是通过基于 Ant 的配置驱动的。主要的 Ant 构建脚本是位于 jboss-3.2/build 目录的 build.xml 文件。其中，该脚本将很多定制的 Ant 任务标记为 buildmagic 构造块。build.xml 文件的目的在于完成目录 jboss-3.2 下不同模块的编译工作，然后将结果集成并创建二进制发布版。发布版可以在 jboss-3.2/build/output 目录找到。上述例子是通过 build.sh 脚本而触发构建过程的。其中，build.sh 对 Ant 进行了包装（wrapper）。当然，如果开发者的环境具备运行 Ant 的条件，也可以通过命令行使用 Ant。

分支的最新源代码，则应该使用 jboss-head 作为模块别名。

7. JBoss CVS 源代码树概述

表 1-2 简要地介绍了 CVS 源代码树各项层目录的主要用途。

表 1-2 JBoss CVS 源代码树的顶层目录描述

目 录	描 述
blocks	未用
build	触发发布版构建过程的主要目录
cluster	群集支持服务源代码模块
common	供其他代码模块使用的实用源代码模块
compatible	当前还处于开发阶段的后向兼容模块
connector	JCA 支持和应用服务器集成源代码模块
console	用于查看 JMX MBean 的 admin 应用
ejb	未用
iiop	RMI/IIOP 传输服务源代码模块
j2ee	标准的 J2EE API 接口和类源代码模块
jboss.net	Web 服务支持源代码模块，为使用 SOAP 调用 EJB 和 MBean 操作提供支持
jmx	JBoss JMX 实现源代码模块
management	JBoss JSR-77 源代码模块
messaging	JBoss JMS 1.0.2b 实现源代码模块
naming	JBoss JNDI 1.2.1 实现源代码模块
security	JBoss 标准的、基于 JAAS 的 J2EE 安全性声明实现
server	EJB 2.0 容器实现相关源代码
system	基于 JMX 微内核的 bootstrap 服务以及标准的部署服务源代码模块
testsuite	JUnit 单元测试源代码模块
thirdparty	JBoss 使用到的第三方二进制 jar 模块
tomcat	Tomcat-4.1.x 嵌入式服务源代码模块
tools	用于 JBoss 构建过程的 jar 文件集合
transaction	JTA 事务管理器
varia	各种实用服务。其中，有些服务将被集成到更高级的模块中，有些则不会

8. 使用 JBossTest 单元测试套件

使用 JBoss 测试套件能够完成 JBoss 安装和构建的更深入测试。JBossTest 套件是面向客户的 JBoss 服务器应用单元测试集合。它同时使用了 Ant 和 JUnit (<http://www.junit.org>) 单元测试框架。另外，JBossTest 套件被 JBoss 开发团队作为质量保证 (QA) 的基准，在测试新功能的同时，也能够防止 bug 的引入。它每天都会运行一次，并将结果传递给开发邮件列表成员。

这些单元测试使用 Ant 运行。测试的源代码位于目录“jboss-3.2/testsuite”。图 1-3 给出了 testsuite CVS 模块的目录结构。

src/main 和 src/resource 为两个主要的源代码分支。其中，src/main 含有用于单元测试的 Java 源代码。src/resources 目录树含有资源文件，比如部署描述符、jar 文件的 manifest

及 Web 内容等。每个单元测试的根包为 `org.jboss.test`。根包以下的各个特定子包通常由组成单元测试类的测试包构成，比如 `security`。由于 Ant 构建脚本在完成单元测试的过程中仅仅寻找 `test` 子包，因此命名规范中必须包含 `test` 子包。如果测试涉及到 EJB，则命名规范中必须为这些 EJB 组件提供包含接口的 `ejb` 子包。单元测试自身需要遵循类文件的命名规范。比如，单元测试类必须命名成：`XXXUnitTest.java`。其中，`XXX` 是待测试的类，或者待测试的功能名。

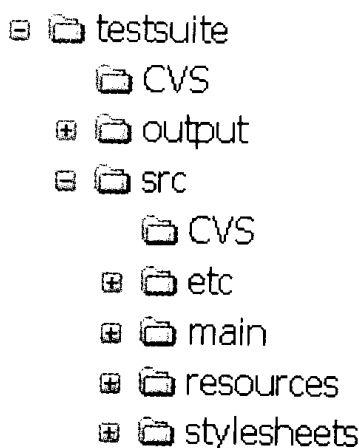


图 1-3 testsuite CVS 模块目录结构

为了能够运行单元测试，开发者必须使用目录 `jboss-3.2/testsuite` 中的 `build` 脚本文件。`build.xml` 文件包含的主要任务如下。

- **tests:** 本任务编译和运行所有的单元测试，并生成 HTML 格式和文本报告格式，最后将它们分别保存在 `testsuite/output/reports/html` 目录和 `testsuite/output/reports/text` 目录中。
- **tests-standard-unit:** 编译所有的单元测试，并运行部分关键的单元测试。如果需要快速地测试服务器以确保不存在重大错误时，这种任务特别适合。
- **test:** 该任务允许运行特定包所包含的所有单元测试。为能够运行该任务，开发者需要在命令行中使用 `-Dtest=package` 指定 `test` 属性和包名。其中，`package` 的取值为位于 `org.jboss.test` 下的目标测试包名。比如，如果运行 `org.jboss.test.naming` 包中所有的单元测试，则可以使用：

```
build.sh -Dtest=naming test
```

- **one-test:** 这种测试任务只允许运行单个单元测试。同样的道理，需要在命令行中使用 `-Dtest=classname` 指定 `test` 属性和目标测试类名（`classname`）。比如，为测试 `org.jboss.test.naming.ENCUnitTestCase` 类，需要使用：

```
build.sh -Dtest=org.jboss.test.naming.test.ENCUnitTestCase one-test
```

- **tests-report:** 该测试任务将位于 `testsuite/output/reports` 目录下现有的 JUnit XML 结果转化为 HTML 格式和文本报告格式。其中，HTML 格式保存在目录 `testsuite/output/reports/html` 下；文本格式位于 `testsuite/output/reports/text` 下。当开

发者手工运行部分测试并需要生成正规的 HTML 报告格式, 以创建一个测试总结的时候, 使用 tests-report 特别合适。

每次完成单元测试时, testsuite/output/reports 目录中都会生成一个或多个 XML 文件, 以保存单个的 JUnit 测试结果。tests-report 任务将这些 XML 文件分别转化为 html 目录下的 HTML 报告格式和 text 目录下的文本报告格式。图 1-4 给出了运行 JBoss-3.2.3 发布版提供的单元测试套件所生成的 HTML 报告实例。

开发者通过 JBoss 发布版的 JBOSS_DIST/docs/tests 目录能够找到测试套件的测试结果。

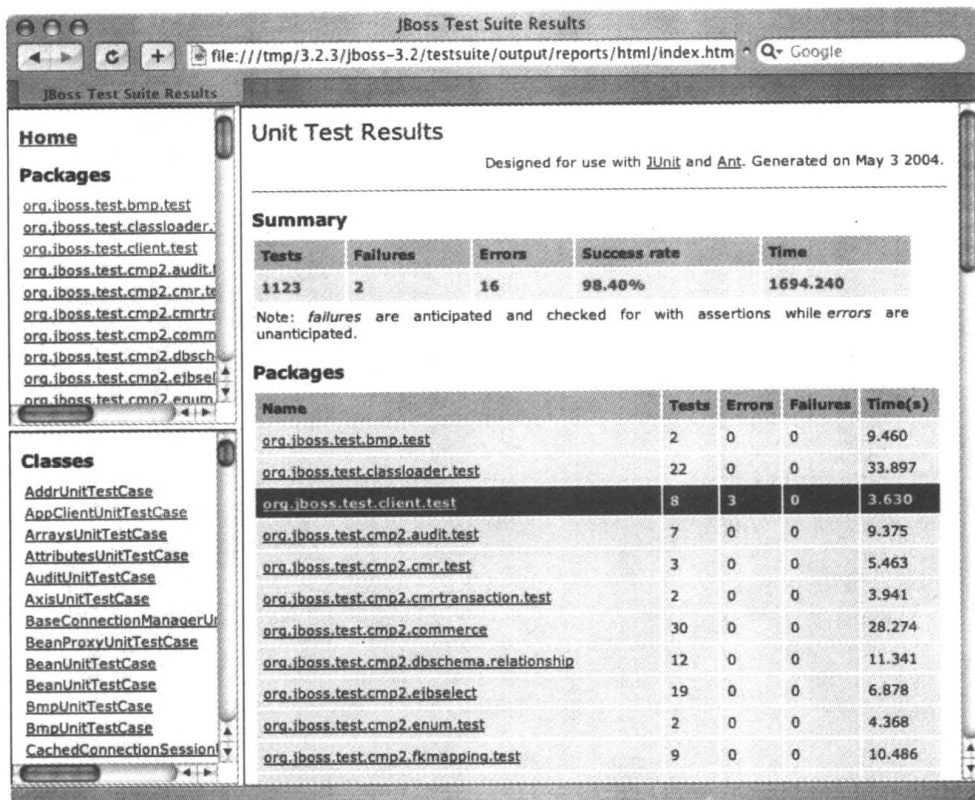


图 1-4 testsuite 生成的 HTML 格式视图实例

第 2 章 JBoss JMX 微内核 (Microkernel)

从零开始、采用模块化开发的 JBoss 服务器和容器，是完全基于组件的插件方式而实现的。而其中最大的功劳应该归于 Java 管理扩展 (Java Management Extension, JMX) API 的使用。通过使用 JMX，工业标准接口能够协助管理 JBoss 服务器和部署在其上的应用程序。JBoss 服务器仍然将易用性作为首要考虑的指标，因此为便于服务器和应用管理，JBoss 服务器 3.x 版引入了全新的模块化、嵌入式设计架构。

模块化架构的高度灵活性为应用开发者提供了如下几方面的优势。首先，能够对紧耦合代码做进一步细化，以满足应用的细粒度控制。比如，如果应用不需要 EJB 挂起功能，只需要将其从服务器去除即可。如果日后需要将相同的应用以应用服务供应商 (Application Service Provider, ASP) 模型部署，则只需要通过基于 Web 部署的控制台将 EJB 挂起功能激活。其次，开发者能够灵活地在容器里面使用他们擅长的对象 - 关系数据库映射 (O/R Mapping) 工具，比如 TopLink。

本章主要介绍 JMX 及它在 JBoss 服务器组件总线中的作用。另外，本章还将介绍 JBoss MBean 服务概念，即如何将生命周期操作添加给 JMX 管理组件。

2.1 JMX 介绍

完整的开源 J2EE 栈 (JBoss) 的成功秘诀在于使用了 JMX (Java 管理扩展)。JMX 是最优秀的软件集成工具。它为集成模块、容器、嵌入式组件提供了通用的方法。图 2-1 展示了 JMX 在作为集成中心 (总线) 并将组件嵌入到 JBoss 服务器中的重要作用。其中，它将声明为 MBean 服务的组件装载到 JBoss 中，从而可以通过 JMX 管理这些组件。

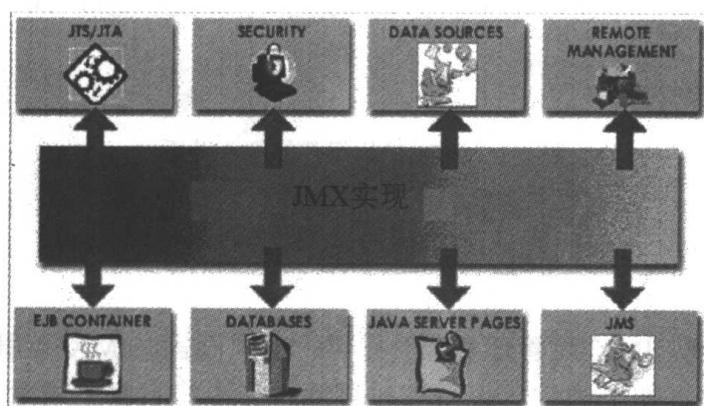


图 2-1 JBoss JMX 集成总线 and 标准 JBoss 组件

在深入讨论 JBoss 如何将 JMX 作为其组件总线之前，本文需要对 JMX 的一些重要方

面给出大体的描述，使得开发者能更好地理解 JBoss。

JMX 组件由 Java 管理扩展装备和代理规范 V1.0 (Java Management Extensions Instrumentation and Agent Specification V1.0) 定义完成。通过 JSR003 Web 页面 <http://jcp.org/aboutJava/communityprocess/final/jsr003/index3.html> 能够找到其详细信息。本节内容主要参考了上述规范，并将重点放在 JBoss 使用到的技术内容上。如果开发者需要更详细的、全面的 JMX 介绍和应用，请参考 Juha Lindfors 著，由 Sams 出版社 2002 年出版发行的《JMX: Managing J2EE with Java Management Extensions》一书 (ISBN: 0-672-32288-9)。

借助于 Java，JMX 提供了管理、监控各种软硬件的标准。进一步而言，JMX 更注重与现有的大量管理标准的集成。图 2-2 描述了 JMX 环境中的组件实例，它描述了 JMX 模型的 3 个层次，以及各个层次之间的相互关系。其中，这 3 个层次的具体含义如下。

- 装配层：受管资源。
- 代理层：用于控制装配层的对象。
- 分布式服务层：描述了管理应用程序如何与代理、受管对象进行交互的机制。

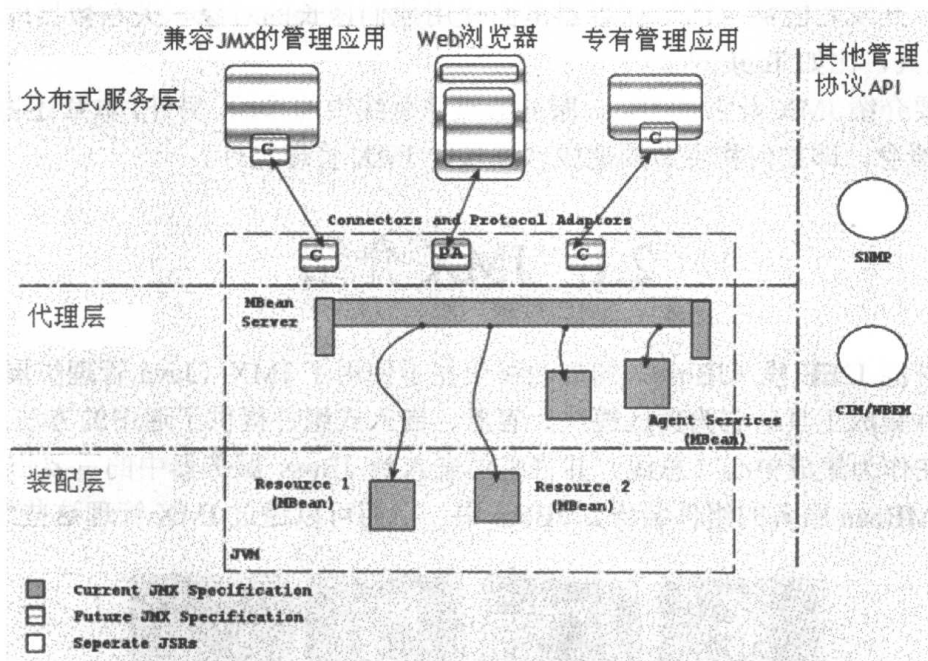


图 2-2 JMX 架构组件的相互关系

2.1.1 装配层

装配层定义了实现 JMX 可管理资源的需求。宏观而言，JMX 可管理资源能够是任何内容，包括应用程序、服务组件、设备等。可管理资源将自身的可管理内容暴露成 Java 对象（或 wrapper），从而使得这些被装备的资源能够被 JMX 兼容应用管理。

开发者使用一个，或多个 Managed Bean（或称为 MBean）为具体资源提供装配。当前，存在以下 4 种不同的 MBean 实现：standard、dynamic、model 和 open。它们的不同点在 2.1.4 节中的“1. Managed Beans 或 MBeans”部分有相关的介绍。

另外, 装配层也提供了通知机制。其目的在于, 当周边环境发生变化时, MBean 之间能够相互通信。这和 JavaBean 属性变化通知机制很类似。通知机制能够用于属性变化通知、状态改变通知, 等等。

2.1.2 代理层

代理层定义了实现代理 (Agent) 的需求。代理的主要职责是为那些注册到代理的受管资源提供控制和管理。在默认情况下, 代理和受管资源在相同的主机上 (可选项)。

代理需求利用装配层定义标准的 MBeanServer 管理代理本身、支持的服务及通信连接器。JBoss 同时提供了 HTML 适配器和 RMI 适配器。

当 JMX 可管理资源驻留的主机上存在 Java 虚拟机时, 代理也可以存放在该主机上。这正是 JBoss 当前使用 MBeanServer 的方式。JMX 代理不需要理会可管理资源的位置。JMX 可管理资源能够使用任何满足它要求的 JMX 代理。

正如下面的内容“分布式服务层”描述的一样, 管理者通过协议适配器 (或连接器) 与 MBean 进行交互。代理不需要理会连接器, 或那些与代理和 MBean 交互的管理应用程序的任何细节信息。

目前, JMX 代理运行在 J2SE 上。一旦 PersonalJava 和 EmbeddedJava 这些平台能够 and Java 2 平台兼容, JMX 规范也将能够运行在它们上。

2.1.3 分布式服务层

JMX 规范的初始版本并没有给出分布式服务层的完整定义, 图 2-2 中带有水平线的组件框暗示了这一点。该层的目的在于, 定义出实现 JMX 管理应用或管理者所需要的接口信息。当前的 JMX 规范中, 分布式服务层需要提供如下几方面的重要功能。

- 为管理应用提供接口。其中, 这些接口能够通过连接器实现与如下内容进行透明地交互: 代理、(由代理负责管理的) JMX 可管理资源。
- 通过把 JMX 代理和其上的 MBean 的语义信息映射到具备丰富数据的协议 (如 HTML 和 SNMP) 块上, 从而能够展示 JMX 代理的管理视图和其上的 MBean。
- 将上游管理平台信息发布到各个 JMX 代理上。
- 将来自不同 JMX 代理的管理信息整合到逻辑视图中。其中, 这些逻辑视图与终端用户的业务操作存在很大关系。
- 提供安全性。

很明显, 分布式服务层组件将实现代理和受管资源的协同网络管理。其中, 这些组件集结后能够提供完善的管理应用。

2.1.4 JMX 组件介绍

本小节内容给出装配层和代理层组件的介绍。

其中, 装配层组件主要如下:

- MBeans (standard、dynamic、open 及 model 类型)。
- 通知模型元素。
- MBean 元数据类。

代理层组件主要有：

- MBean 服务器。
- 代理服务。

1. Managed Beans 或 MBeans

MBean 是 Java 对象，它实现了标准的 MBean 接口，并遵循相关的设计模式实现。对于资源的 MBean 而言，它为控制其本身的管理应用暴露了所需的信息和操作。

MBean 的管理接口需要负责如下一些内容：

- 通过名字访问属性值。
- 提供被允许调用的操作或功能。
- 能够发出通知或事件。
- 提供 MBean Java 类的构建器。

JMX 定义了 4 种 MBean 以支持不同的装配需求。

- 标准 MBean：它使用简单 JavaBean 风格的命名约定，并且以静态方式定义管理接口。当前，JBoss 服务器主要使用这种 MBean 类型。
- 动态 MBean：它必须实现 `javax.management.DynamicMBean` 接口。这种 MBean 只有在运行时才会暴露其管理接口，从而为初始化组件提供了最高程度的灵活性。JBoss 在如下场合才会使用动态 MBean，即直到运行时才知道组件受管的目的地。
- 开放 MBean：它是对动态 MBean 的扩展。为实现通用管理性，开放 MBean 依赖于基本的、自描述及用户友好的数据类型。从 JBoss 3.0.6 和 3.2 开始，JBoss 才实现了 JMX 1.1 版本的开放 MBean。
- 模型 MBean：它也是对动态 MBean 的扩展。模型 MBean 必须实现 `javax.management.modelmbean.ModelMBean` 接口。模型 MBean 通过提供默认行为简化了资源的装配。尽管目前的 JBoss 3.2.0 发布版的核心服务并没有使用任何模型 MBean，但它还是提供了模型 MBean 实现，即 XMBean。

本书在讨论如何使用自定义服务扩展 JBoss 时，将会给出标准 MBean 和模型 MBean 实例。

2. 通知模型

JMX 通知扩展了 Java 事件模型。MBeanServer 和 MBean 都能够发出通知以提供信息。JMX 规范在 `javax.management` 包中定义了 `Notification` 事件对象、`NotificationBroadcaster` 事件发送者接口及 `NotificationListener` 事件监听者接口。另外，JMX 规范还定义了相关的 MBeanServer 操作，用于注册事件监听器。

3. MBean 元数据类

为描述 MBean 的管理接口, JMX 规范定义了一套元数据类。通过查询注册了 MBean 的 MBean 服务器, 开发者能够获得上述 4 种 MBean 类型中任何一种 MBean 的元数据视图。其中, 元数据类涉及到 MBean 的属性、操作、通知及构建器。对于上述各种涉及到的内容, 元数据必须包括名字、描述及具体特性。比如, 属性有一种特性指是否可读或可写, 或者两者都可。对于操作而言, 元数据包含了参数和返回类型的语法规则 (signature)。

各种类型的 MBean 都扩展了元数据类, 从而能够提供各 MBean 所需的扩展信息。基于这种通用的继承特性, 使得无论 MBean 的具体类型是什么, 开发者都可以获得其标准信息。另外, 管理应用知道如何访问特定 MBean 类型的扩展信息。

4. MBean 服务器

MBean 服务器是代理层的重要组件。通过 `javax.management.MBeanServer` 实例能够获悉它提供的具体功能。MBeanServer 是 MBean 服务的注册中心, 即管理应用程序通过 MBeanServer 能够获得已注册 MBean 的管理接口。MBean 服务器从不直接暴露 MBean 对象本身, 而是暴露其管理接口。其中, 这些管理接口是通过元数据和 MBeanServer 接口的操作获得的, 这种方式使得管理应用程序和受管 MBean 处于松耦合状态。

实例化 MBean、并注册到 MBeanServer 的方式有如下几种:

- 其他的 Mbean。
- 代理本身。
- 远程管理应用程序 (通过分布式服务)。

在注册 MBean 的过程中, 开发者必须为 MBean 提供惟一的对象名。然后, 管理应用程序便能够借助于对象名 (即, 惟一 handle) 定位到具体的 MBean, 以完成管理操作。

MBeanServer 服务器对 MBean 的操作类型主要有:

- 发现 MBean 的管理接口。
- 读取和写入属性值。
- 调用 MBean 定义的操作。
- 注册通知事件。
- 通过对象名或属性值查询 Mbean。

协议适配器和连接器需要从代理服务所运行的 JVM 之外访问 MBeanServer。通过连接到 MBeanServer, 适配器借助于各自的协议能够获得 MBeanServer 上注册的 MBean 视图。比如, HTML 适配器允许使用 Web 浏览器查看和编辑 MBean。正如图 2-2 描述的一样, 当前版本的 JMX 规范并没有定义协议适配器。JMX 规范的后续版本将会重点关注远程访问协议以标准方式访问 MBeanServer 的需求。

连接器是供管理应用程序使用的接口, 它主要是为实现与底层通信协议无关的方式访问 MBeanServer 而提供的通用 API。每种连接器类型针对不同的协议提供了相同的远程接口, 这使得远程管理应用程序可以不管具体的协议类型, 透明地通过网络连接到代理。远程管理接口规范将在 JMX 规范的后续版本中给出。

适配器和连接器使得所有的 MBean 服务器操作能够供远程管理应用使用。如果需要从运行代理的 JVM 外部管理它, 则开发者必须提供协议适配器或连接器。目前, JBoss 包

含了自定义 HTML 适配器和 RMI 适配器实现。

5. 代理服务

JMX 代理服务指那些能够对注册在 MBean 服务器上的 MBean 进行标准操作的对象。这种支持管理服务的引入有助于构建更加有力的管理方案。在通常情况下，由于代理服务本身也是 MBean，因此能够通过 MBean 服务器控制代理及其功能。JMX 规范定义了如下的代理服务。

- 动态类装载 MLet（管理 Java 程序）服务：该服务使得从目的网络位置获得并实例化新类和本地库成为可能。
- 监控服务：观察 MBean 属性的数值或字符串值，并将这些变化通知给其他目标对象。
- 定时服务：提供基于定时通知或重复周期通知的定时机制。
- 关系服务：定义 MBean 之间的相互关系，并确保这些关系的一致性。

任何 JMX 兼容实现都将提供上述所有代理服务。当前，JBoss 并不依赖于这些标准代理服务。

2.2 JBoss JMX 实现架构

2.2.1 JBoss 类装载器架构

JBoss 3.x 实现了一种新的类装载架构，即允许类跨部署单元使用。在 JBoss 2.x 中，很难实现 MBean 服务和动态部署的 J2EE 组件进行交互，并且 MBean 本身不具有热部署能力。在 JBoss 3.x 中，任何东西都是热部署的，因为新的部署架构和类装载架构使得它们成为可能。本文在深入讨论具体的 JBoss 类装载模型之前，将给出 Java 的类型系统特性及类装载器介绍。

2.2.2 类装载和 Java 中的类型

类装载是所有服务器架构的基础组成部分。具体服务及其支持服务的类必须装载到服务器框架中。Java 的强类型化（typed）特性使得类装载易于出现问题。大部分开发者知道，Java 中类的类型是由类的全限定（fully qualified）名决定的。从 Java 1.2 开始，用于定义类的 `java.lang.ClassLoader` 也能够决定类的类型。这样做的理由在于确保从特定位置装载类的环境是类型安全的（type-safe）。Vijay Saraswat 于 1997 年发表的论文“Java is not type-safe¹”证明了 Java 不是类型安全的。这使得应用能够使用先前装载的其他类实现版本而愚弄 JVM，从而访问到 Java 不应该访问的类方法和成员。这种对类型系统的欺骗的根源在于这些应用引入了那些能够跨越正常委派模型的类装载器。类装载器使用委派模型

¹ 译者注：论文全文请参考 <http://matrix.research.att.com/vj/bug.html>。

搜索类和资源。在通常情况下，类装载器实例都有与之关联的双亲类装载器，它可能是在创建类装载器时显式设置的；如果没有给出双亲类装载器，则由 JVM 指定。如果需要寻找类或资源，则在类装载器自身去寻找它们之前，通常情况下都会将该搜索工作委派给其双亲类装载器。JVM 有根类装载器，称为引导 (bootstrap) 类装载器。引导类装载器没有双亲类装载器，但能够成为其他类装载器实例的双亲。

为了解决类型安全问题，Java 类型系统除了通过类名完整地定义类型外，还引入了用于定义类的类装载器。其内容请参考由 Sheng Liang 和 Gilad Bracha 完成的论文《Dynamic Class Loading in the Java Virtual Machine》。同时，通过 Web 地址 <http://java.sun.com/people/sl/papers/oops1a98.ps.gz>，开发者能够获得其原文。另外，在动态环境中，比如应用服务器（尤其是支持热部署的 JBoss），动态类装载方式有了更深入的发展。其中，ClassCastException、LinkageError 及 IllegalAccessError 更能展示出静态类装载上下文中所看不到的场景。本文接下来仔细研究上述各种异常的具体含义和发生方式。

1. ClassCastException——我不是你的类型

无论在什么情况下，只要将实例造型 (cast) 作为其不兼容的类型，系统就会抛出 java.lang.ClassCastException 异常。比如，某简单实例如下：将 java.net.URL 放入 java.util.ArrayList 后，试图获得 java.lang.String 的代码和异常信息如下：

```
ArrayList array = new ArrayList();
array.add(new URL("file:/tmp"));
String url = (String) array.get(0);

java.lang.ClassCastException: java.net.URL
at org.jboss.chap2.ex0.ExCCEa.main(Ex1CCE.java:16)
```

开发者可以从 ClassCastException 看出，将 array 元素造型作为 String 类型失败，因为该元素的实际类型为 java.net.URL。当然，开发者对这种试验性的场景不会感兴趣。让我们考虑另一种场景：不同的 URLClassLoader 装载了相同的 jar 文件。从字节码角度考虑，尽管通过不同 URLClassLoader 装载的类是完全相同的，但它们被 Java 类型系统看做完全不同的类型。列表 2-1、列表 2-2 和列表 2-3 给出了相应的代码实例。

列表 2-1 用于证明由于不同类装载器而触发 ClassCastException 的 ExCCEc 类

```
1 package org.jboss.chap2.ex0;
2
3 import java.io.File;
4 import java.net.URL;
5 import java.net.URLClassLoader;
6 import java.lang.reflect.Method;
7
8 import org.apache.log4j.Logger;
9
10 import org.jboss.util.ChapterExRepository;
11 import org.jboss.util.Debug;
12
```

```
13 /** An example of a ClassCastException that results from classes loaded through
14  * different class loaders.
15  * @author Scott.Stark@jboss.org
16  * @version $Revision:$
17  */
18 public class ExCCEc
19 {
20     public static void main(String[] args) throws Exception
21     {
22         ChapterExRepository.init(ExCCEc.class);
23
24         String chapDir = System.getProperty("chapter.dir");
25         Logger ucl0Log = Logger.getLogger("UCL0");
26         File jar0 = new File(chapDir+"/j0.jar");
27         ucl0Log.info("jar0 path: "+jar0.toString());
28         URL[] cp0 = {jar0.toURL()};
29         URLClassLoader ucl0 = new URLClassLoader(cp0);
30         Thread.currentThread().setContextClassLoader(ucl0);
31         Class objClass = ucl0.loadClass("org.jboss.chap2.ex0.ExObj");
32         StringBuffer buffer = new StringBuffer("ExObj Info");
33         Debug.displayClassInfo(objClass, buffer, false);
34         ucl0Log.info(buffer.toString());
35         Object value = objClass.newInstance();
36
37         File jar1 = new File(chapDir+"/j0.jar");
38         Logger ucl1Log = Logger.getLogger("UCL1");
39         ucl1Log.info("jar1 path: "+jar1.toString());
40         URL[] cp1 = {jar1.toURL()};
41         URLClassLoader ucl1 = new URLClassLoader(cp1);
42         Thread.currentThread().setContextClassLoader(ucl1);
43         Class ctxClass2 = ucl1.loadClass("org.jboss.chap2.ex0.ExCtx");
44         buffer.setLength(0);
45         buffer.append("ExCtx Info");
46         Debug.displayClassInfo(ctxClass2, buffer, false);
47         ucl1Log.info(buffer.toString());
48         Object ctx2 = ctxClass2.newInstance();
49
50         try
51         {
52             Class[] types = {Object.class};
53             Method useValue = ctxClass2.getMethod("useValue", types);
54             Object[] margs = {value};
55             useValue.invoke(ctx2, margs);
56         }
57         catch(Exception e)
58         {
```

```
59         ucllLog.error("Failed to invoke ExCtx.useValue", e);
60         throw e;
61     }
62 }
63 }
```

列表 2-2 实例使用到的 ExCtx 类

```
64 package org.jboss.chap2.ex0;
65
66 import java.io.IOException;
67
68 import org.apache.log4j.Logger;
69
70 import org.jboss.util.Debug;
71
72 /** A classes used to demonstrate various class loading issues
73  * @author Scott.Stark@jboss.org
74  * @version $Revision: 1.2 $
75  */
76 public class ExCtx
77 {
78     ExObj value;
79
80     public ExCtx() throws IOException
81     {
82         value = new ExObj();
83         Logger log = Logger.getLogger(ExCtx.class);
84         StringBuffer buffer = new StringBuffer("ctor.ExObj");
85         Debug.displayClassInfo(value.getClass(), buffer, false);
86         log.info(buffer.toString());
87         ExObj2 obj2 = value.ivar;
88         buffer.setLength(0);
89         buffer = new StringBuffer("ctor.ExObj.ivar");
90         Debug.displayClassInfo(obj2.getClass(), buffer, false);
91         log.info(buffer.toString());
92     }
93     public Object getValue()
94     {
95         return value;
96     }
97     public void useValue(Object obj) throws Exception
98     {
99         Logger log = Logger.getLogger(ExCtx.class);
100         StringBuffer buffer = new StringBuffer("useValue2.arg class");
101         Debug.displayClassInfo(obj.getClass(), buffer, false);
```

```

102     log.info(buffer.toString());
103     buffer.setLength(0);
104     buffer.append("useValue2.ExObj class");
105     Debug.displayClassInfo(ExObj.class, buffer, false);
106     log.info(buffer.toString());
107     ExObj ex = (ExObj) obj;
108 }
109 void pkgUseValue(Object obj) throws Exception
110 {
111     Logger log = Logger.getLogger(ExCtx.class);
112     log.info("In pkgUseValue");
113 }

114 }
115

```

列表 2-3 实例使用到的 ExObj 和 ExObj2 类

```

package org.jboss.chap2.ex0;

import java.io.Serializable;

/**
 * @author Scott.Stark@jboss.org
 * @version $Revision:$
 */
public class ExObj implements Serializable
{
    public ExObj2 ivar = new ExObj2();
}

-----
package org.jboss.chap2.ex0;

import java.io.Serializable;

/**
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.1$
 */

public class ExObj2 implements Serializable
{
}

```

ExCCEc.main 方法使用反射将应用程序的类装载器与 URLClassLoader ucl0 和 ucl1 各自装载的类隔离开，而这些类都是从 output/chap2/j0.jar 文件中装载的。j0.jar 的具体内容

如下:

```
[nr@toki examples]$ jar -tf output/chap2/j0.jar
...
org/jboss/chap2/ex0/ExCtx.class
org/jboss/chap2/ex0/ExObj.class
org/jboss/chap2/ex0/ExObj2.class
```

本文将通过运行实例来证明 `ClassCastException` 是如何出现的, 然后再来研究问题所在。请参考附录 C 提供的本书附带实例的安装指南。从光盘目录使用如下命令运行该实例:

```
[nr@toki examples]$ ant -Dchap=chap2 -Dex=0c run-example
Buildfile: build.xml
...
[java] [ERROR,UCL1] Failed to invoke ExCtx.useValue
[java] java.lang.reflect.InvocationTargetException
[java] at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
[java] at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
[java] at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
[java] at java.lang.reflect.Method.invoke(Method.java:324)
[java] at org.jboss.chap2.ex0.ExCCEc.main(ExCCEc.java:58)
[java] Caused by: java.lang.ClassCastException
[java] at org.jboss.chap2.ex0.ExCtx.useValue(ExCtx.java:44)
[java] ... 5 more
[java] Java Result: 1
```

上述给出的只是异常信息, 通过 `logs/chap2-ex0c.log` 文件能够找到完整的输出信息。其中, `ExCCEc.java` 代码使用 `URLClassLoader ucl1` 装载了 `ExCtx` 类, 然后实例化该类 (第 37~48 行)。最后, 调用了 `ExCtx.useValue(Object)` 方法 (第 55 行)。另外, 传进来的 `ExObj` 实例 (译者注: `value`) 是借助于 `URLClassLoader ucl0` (第 25~35 行) 装载的。当 `ExCtx.useValue` 代码试图将传入参数造型作为 `ExObj` 对象时, 应用抛出了异常。为更好地理解失败的原因, 列表 2-4 给出了摘自 `chap2-ex0c.log` 文件的调试输出信息。

列表 2-4 用于 `ExObj` 类的 `chap2-ex0c.log` 调试输出

```
[INFO,UCL0] ExObj Info
org.jboss.chap2.ex0.ExObj(113fe2).ClassLoader=java.net.URLClassLoader@6e3914
..java.net.URLClassLoader@6e3914
....file:/C:/Scott/JBoss/Books/AdminDevel/education/books/admin-devel/examples/output/
chap2/j0.jar
++++CodeSource: (file:/C:/Scott/JBoss/Books/AdminDevel/education/books/admin-devel/
examples/output/chap2/j0.jar <no certificates>)
Implemented Interfaces:
++interface java.io.Serializable(7934ad)
++++ClassLoader: null
```

```
++++Null CodeSource
[INFO,ExCtx] useValue2.ExObj class
org.jboss.chap2.ex0.ExObj(415de6).ClassLoader=java.net.URLClassLoader@30e280
..java.net.URLClassLoader@30e280
....file:/C:/Scott/JBoss/Books/AdminDev/education/books/admin-dev/examples/output/
chap2/j0.jar
++++CodeSource: (file:/C:/Scott/JBoss/Books/AdminDev/education/books/admin-dev/
examples/output/chap2/j0.jar <no certificates>)
Implemented Interfaces:
++interface java.io.Serializable(7934ad)
++++ClassLoader: null
++++Null CodeSource
```

带有前缀[INFO,UCL0]的第一条输出表明，ExCCEc.java 中第 31 行装载的 ExObj 类的 Hash 码为 113fe2，与之关联的 URLClassLoader 实例 (ucl0) 的 Hash 码为 6e3941。传递给 ExCtx.useValue 方法的实例就是基于该 ExObj 类创建的。带有前缀[INFO,ExCtx]的第二条输出表明，ExCtx.userValue 方法上下文中的 ExObj 实例的 Hash 码为 415de6，与之关联的 URLClassLoader 实例 (ucl1) 的 Hash 码为 30e280。尽管这两个 ExObj 类是来自相同 j0.jar 中的同一字节码文件，但结果是两者的散列码和 URLClassLoader 都不同。因此，试图将不同范围的 ExObj 实例进行造型转换，将会抛出 ClassCastException 异常。

若开发者重新部署某应用，与此同时有其他的应用引用了该应用的类，则此时 ClassCastException 异常经常出现，如单独的 Web 应用 (.war) 访问 EJB。如果重新部署了某应用，所有依赖于它的应用必须刷新 (flush) 它们的类引用。一般情况下，这要求依赖的应用程序能够自动重新部署。

当出现了重新部署的情况时，为实现独立部署应用之间的交互，一种替代方式是通过配置 EJB 层以隔离不同的部署应用。其具体办法是，将 EJB 层默认的引用访问语义修改为传值访问语义。因为基于引用访问语义时，JBoss 设定部署在其上的组件默认运行在同一 JVM 中。

2. IllegalAccessException——做不应该做的

当试图访问可见性限定符 (visibility qualifier) 不允许访问的方法或成员时，应用会抛出 java.lang.IllegalAccessException 异常。常见的例子有：试图访问私有或受保护的方法和实例变量。再比如，存在某例子，从表面上看是从正确的包访问私有受保护的方法或成员，但实际上访问者和被访问者是由不同类装载器装载的。列表 2-5 给出了代码实例。

列表 2-5 由于不同类装载器引起 IllegalAccessException 的 ExIAEd 类

```
116 package org.jboss.chap2.ex0;
117
118 import java.io.File;
119 import java.net.URL;
120 import java.net.URLClassLoader;
121 import java.lang.reflect.Method;
122
```

```
123 import org.apache.log4j.Logger;
124
125 import org.jboss.util.ChapterExRepository;
126 import org.jboss.util.Debug;
127
128 /** An example of IllegalAccessExceptions due to classes loaded by two class
129  * loaders.
130  * @author Scott.Stark@jboss.org
131  * @version $Revision: 1.2$
132  */
133 public class ExIAEd
134 {
135     public static void main(String[] args) throws Exception
136     {
137         ChapterExRepository.init(ExIAEd.class);
138
139         String chapDir = System.getProperty("chapter.dir");
140         Logger ucl0Log = Logger.getLogger("UCL0");
141         File jar0 = new File(chapDir+"/j0.jar");
142         ucl0Log.info("jar0 path: "+jar0.toString());
143         URL[] cp0 = {jar0.toURL()};
144         URLClassLoader ucl0 = new URLClassLoader(cp0);
145         Thread.currentThread().setContextClassLoader(ucl0);
146
147         StringBuffer buffer = new StringBuffer("ExIAEd Info");
148         Debug.displayClassInfo(ExIAEd.class, buffer, false);
149         ucl0Log.info(buffer.toString());
150
151         Class ctxClass1 = ucl0.loadClass("org.jboss.chap2.ex0.ExCtx");
152         buffer.setLength(0);
153         buffer.append("ExCtx Info");
154         Debug.displayClassInfo(ctxClass1, buffer, false);
155         ucl0Log.info(buffer.toString());
156         Object ctx0 = ctxClass1.newInstance();
157
158         try
159         {
160             Class[] types = {Object.class};
161             Method useValue = ctxClass1.getDeclaredMethod("pkgUseValue", types);
162             Object[] margs = {null};
163             useValue.invoke(ctx0, margs);
164         }
165         catch(Exception e)
166         {
167             ucl0Log.error("Failed to invoke ExCtx.pkgUseValue", e);
168         }
169     }
170 }
```

```
169 }  
170 }
```

应用类装载器将 ExIAEd 类装载到应用中，但 ExCtx 类是 ExIAEd.main 方法通过反射并借助于 URLClassLoader ucl0 装载的。这里将运行上述实例来证明如何抛出 IllegalAccessException 异常，并找到问题的原因。通过如下命令运行实例：

```
[orb@toki examples]$ ant -Dchap=chap2 -Dex=0d run-example  
Buildfile: build.xml  
...  
[java] [ERROR,UCL0] Failed to invoke ExCtx.pkgUseValue  
[java] java.lang.IllegalAccessException: Class org.jboss.chap2.ex0.ExIAEd can not access a  
member of class org.jboss.chap2.ex0.ExCtx with modifiers ""  
[java] at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:57)  
[java] at java.lang.reflect.Method.invoke(Method.java:317)  
[java] at org.jboss.chap2.ex0.ExIAEd.main(ExIAEd.java:48)
```

上述那条大块输出信息展示了 IllegalAccessException 异常的抛出。完整的输出信息能在 logs/chap2-ex0d.log 文件中找到。ExIAEd.java 中的第 48 行，程序借助于反射调用了 ExCtx.pkgUserValue(Object) 方法。其中，pkgUseValue 方法具有包受保护访问限定，尽管调用类 ExIAEd 和方法被调用的 ExCtx 类都位于包 org.jboss.chap2.ex0 中，但由于这两个类被不同类装载器装载，因此调用无效。通过查看 chap2-ex0d.log 文件，能够找到重要的几行输出信息：

```
[INFO,UCL0] ExIAEd Info  
org.jboss.chap2.ex0.ExIAEd(65855a).ClassLoader=sun.misc.Launcher$AppClassLoader@3f52a5  
..sun.misc.Launcher$AppClassLoader@3f52a5  
...  
[INFO,UCL0] ExCtx Info  
org.jboss.chap2.ex0.ExCtx(70eed6).ClassLoader=java.net.URLClassLoader@113fe2  
..java.net.URLClassLoader@113fe2  
...
```

通过上述内容可以看出，ExIAEd 类是由默认的应用类装载器实例，即 sun.misc.Launcher\$AppClassLoader@3f52a2 装载的，而 ExCtx 类是通过类装载器实例 java.net.URLClassLoader@113FE2 装载的。由于不同的类装载器装载了这些类，访问包受保护方法便成了安全侵害。因此，类的全限定名和类装载器不仅能够决定类的类型，包范围（package scope）也是如此。

比如，在实际工作中，如果将相同的类放置在两个不同的 SAR（译者注：JBoss Service Archive，JBoss 服务存档）部署文件中就会出现这种异常。如果 SAR 部署文件中的类有包受保护关系，则使用 SAR 服务的用户可能从某个 SAR 装载某类，而后又去另一个 SAR 文件中装载另一个类。如果这两个类有受保护关系，则会抛出 IllegalAccessError 异常。解决的办法可以是：开发者需要将 SAR 引用的类单独放在某个 jar 中，或者将这些 SAR 合并成单个部署文件。因此，有可能是单个的 SAR，或者将这两个 SAR 包含在 EAR（译者注：Enterprise Application Archive，企业应用存档）中。

3. LinkageError——确保你就是声称的你

为解决早期 Java 虚拟机中的类型安全问题, Java 语言规范第 1.2 版引入了装载约束。当某 X 类涉及到多个类装载器时, 为保证它的一致性, 通过装载约束能够在类装载器范围的上下文内验证这种类型是否是预期的。由于 Java 允许用户自定义类装载器, 同时 `LinkageError` 还扩展了 `ClassCastException` 类, 而且装载和使用类的过程中都需要完成装载约束, 因此装载约束是很重要的。

为了能够理解装载约束的内容和它是如何保证类型安全的, 下面首先介绍 Liang 和 Bracha 论文中的术语和例子。当前存在两种类装载器, 即初始和定义类装载器。其中, 初始类装载器是通过调用 `ClassLoader.loadClass` 方法以初始装载命名类的类装载器。而定义类装载器是通过调用 `ClassLoader.defineClass` 方法将类字节码转换成类实例的类装载器。类的完整表达式为:

$$\langle C, L_d \rangle^{L_i}$$

其中, C 为全限定类名, L_d 为定义类装载器, L_i 为初始类装载器。当初始类装载器不重要时, 类型可以表示为 $\langle C, L_d \rangle$; 当定义类装载器不重要时, 类型可以表示为 C^{L_i} 。对于第二种情形, 并不是不存在定义类装载器, 而只是标识它并不重要而已。同时, $\langle C, L_d \rangle$ 完全定义了类型。仅仅在验证装载约束时, 初始装载器才会对 $\langle C, L_d \rangle$ 产生关联。接下来, 请开发者浏览列表 2-6。

列表 2-6 用于证明装载约束的类

```
class <C, L1>
{
    void f()
    {
        <Spoofed, L1>L1 x = <Delegated, L2>L1 . g();
        x.secret_value = 1 ; // Should not be allowed
    }
}
```

```
-----
class <Delegated, L2>
{
    static <Spoofed, L2>L3 g() {...}
}
```

```
-----
class <Spoofed, L1>
{
    public int secret_value;
}
```

```
-----
class <Spoofed, L2>
{
    private int secret_value;
```


其中：类 *C* 由 L_1 定义，因此 L_1 用于初始装载类 *Spoofed* 及 *C.f()* 引用的 *Delegated* 类。*Spoofed* 由 L_1 定义，*Delegated* 由 L_2 定义（由 L_1 委派给 L_2 ）。既然 *Delegated* 由 L_1 定义，则 L_2 将负责 *Delegated.g()* 方法上下文中 *Spoofed* 类的初始装入。本实例中的 L_1 和 L_2 定义了不同版本的 *Spoofed*（列表 2-6 后续内容指出了）。另外，由于 1.1 及先前版本的 JVM 并没有将类的全限定名和定义类装载器，这两者一同决定类的类型，因此既然 *C.f()* 认定 *x* 是 $\langle \text{Spoofed}, L_1 \rangle$ 实例，所以 *x* 能够访问 *Delegated.g()* 返回 $\langle \text{Spoofed}, L_2 \rangle$ 中的私有成员 *secret_value*。

从 Java 1.2 以后，对于从不同定义类装载器中使用类型的情形，通过生成装载约束以验证类型一致性来解决其存在的上述问题。对于列表 2-6 所示的实例，当执行到 *C.f()* 第一行时，JVM 生成 $\text{Spoofed}^{L_1} = \text{Spoofed}^{L_2}$ 约束，从而确保 L_1 和 L_2 初始装载的类型 *Spoofed* 都是相同的。问题的根本不在于 L_1 或 L_2 ，或其他类装载器定义 *Spoofed*，而是无论使用任何类装载器完成初始装载，仅仅能够存在单个 *Spoofed* 类定义。如果 L_1 或 L_2 已经定义了不同版本的 *Spoofed*，只要触发该装载约束检查，JVM 会立即抛出 *LinkageError*。否则，该记录将约束，在执行 *Delegated.g()* 时试图装入重复版本的 *Spoofed* 也将抛出 *LinkageError*。

接下来，本文结合具体实例来体会 *LinkageError* 是如何出现的。列表 2-7 给出了实例主类和使用的自定义类装载器。

列表 2-7 LinkageError 具体实例

```
171 package org.jboss.chap2.ex0;
172
173 import java.io.File;
174 import java.net.URL;
175
176 import org.apache.log4j.Logger;
177 import org.jboss.util.ChapterExRepository;
178 import org.jboss.util.Debug;
179
180 /** An example of a LinkageError due to classes being defined by more than
181  * one class loader in a non-standard class loading environment.
182  * @author Scott.Stark@jboss.org
183  * @version $Revision: 1.1$
184  */
185 public class ExLE
186 {
187     public static void main(String[] args) throws Exception
188     {
189         ChapterExRepository.init(ExLE.class);
190
191         String chapDir = System.getProperty("chapter.dir");
192         Logger ucl0Log = Logger.getLogger("UCL0");
```

```
193     File jar0 = new File(chapDir+"/j0.jar");
194     ucl0Log.info("jar0 path: "+jar0.toString());
195     URL[] cp0 = {jar0.toURL()};
196     ExURLClassLoader ucl0 = new ExURLClassLoader(cp0);
197     Thread.currentThread().setContextClassLoader(ucl0);
198     Class ctxClass1 = ucl0.loadClass("org.jboss.chap2.ex0.ExCtx");
199     Class obj2Class1 = ucl0.loadClass("org.jboss.chap2.ex0.ExObj2");
200     StringBuffer buffer = new StringBuffer("ExCtx Info");
201     Debug.displayClassInfo(ctxClass1, buffer, false);
202     ucl0Log.info(buffer.toString());
203     buffer.setLength(0);
204     buffer.append("ExObj2 Info, UCL0");
205     Debug.displayClassInfo(obj2Class1, buffer, false);
206     ucl0Log.info(buffer.toString());
207
208     File jar1 = new File(chapDir+"/j1.jar");
209     Logger ucl1Log = Logger.getLogger("UCL1");
210     ucl1Log.info("jar1 path: "+jar1.toString());
211     URL[] cp1 = {jar1.toURL()};
212     ExURLClassLoader ucl1 = new ExURLClassLoader(cp1);
213     Class obj2Class2 = ucl1.loadClass("org.jboss.chap2.ex0.ExObj2");
214     buffer.setLength(0);
215     buffer.append("ExObj2 Info, UCL1");
216     Debug.displayClassInfo(obj2Class2, buffer, false);
217     ucl1Log.info(buffer.toString());
218
219     ucl0.setDelegate(ucl1);
220     try
221     {
222         ucl0Log.info("Try ExCtx.newInstance()");
223         Object ctx0 = ctxClass1.newInstance();
224         ucl0Log.info("ExCtx.ctor succeeded, ctx0: "+ctx0);
225     }
226     catch(Throwable e)
227     {
228         ucl0Log.error("ExCtx.ctor failed", e);
229     }
230 }
231 }
-----
232 package org.jboss.chap2.ex0;
233
234 import java.net.URLClassLoader;
235 import java.net.URL;
236
237 import org.apache.log4j.Logger;
```

```
238
239 /** A custom class loader that overrides the standard parent delegation model
240  * @author Scott.Stark@jboss.org
241  * @version $Revision:$
242  */
243 public class ExURLClassLoader extends URLClassLoader
244 {
245
246     private static Logger log = Logger.getLogger(ExURLClassLoader.class);
247     private ExURLClassLoader delegate;
248
249     public ExURLClassLoader(URL[] urls)
250     {
251         super(urls);
252     }
253
254     void setDelegate(ExURLClassLoader delegate)
255     {
256         this.delegate = delegate;
257     }
258
259     protected synchronized Class loadClass(String name, boolean resolve)
260         throws ClassNotFoundException
261     {
262         Class clazz = null;
263         if( delegate != null )
264         {
265             log.debug(Integer.toHexString(hashCode())+"; Asking delegate to loadClass:
"+name);
266             clazz = delegate.loadClass(name, resolve);
267             log.debug(Integer.toHexString(hashCode())+"; Delegate returned: "+clazz);
268         }
269         else
270         {
271             log.debug(Integer.toHexString(hashCode())+"; Asking super to loadClass:
"+name);
272             clazz = super.loadClass(name, resolve);
273             log.debug(Integer.toHexString(hashCode())+"; Super returned: "+clazz);
274         }
275         return clazz;
276     }
277
278     protected Class findClass(String name)
279         throws ClassNotFoundException
280     {
281         Class clazz = null;
```

```

282     log.debug(Integer.toHexString(hashCode())+"; Asking super to findClass:
"+name);
283     clazz = super.findClass(name);
284     log.debug(Integer.toHexString(hashCode())+"; Super returned: "+clazz);
285     return clazz;
286 }
287 }

```

本实例中最重要的组件是 URLClassLoader 的子类 Ex0URLClassLoader。该类装载器实现重载了默认的双亲委派模型,使得 ucl0 和 ucl1 实例都能装载 ExObj2 类,并从 ucl0 到 ucl1 设置了委派关系。在 ExLE.mai 方法中,第 28~29 行,Ex0URLClassLoader 类型的 ucl0 装载了 ExCtx 和 ExObj2 类。第 43 行,Ex0URLClassLoader 类型的 ucl1 再次装载了 ExObj2 类。这时,两个类装载器都定义了 ExObj2 类。第 49 行,借助于 ucl0.setDelegate(ucl1)方法访问使得从 ucl0 到 ucl1 设置了委派关系。最后,第 53 行,借助于 ucl0 装载的类(译者注:ctxClass1)创建了 ExCtx 实例。其中,这里的 ExCtx 类和列表 2-2 中的一样,其构建器如下:

```

288 public ExCtx() throws IOException
289 {
290     value = new ExObj();
291     Logger log = Logger.getLogger(ExCtx.class);
292     StringBuffer buffer = new StringBuffer("ctor.ExObj");
293     Debug.displayClassInfo(value.getClass(), buffer, false);
294     log.info(buffer.toString());
295     ExObj2 obj2 = value.ivar;
296     buffer.setLength(0);
297     buffer = new StringBuffer("ctor.ExObj.ivar");
298     Debug.displayClassInfo(obj2.getClass(), buffer, false);
299     log.info(buffer.toString());
300 }

```

现在,既然 ucl0 类装载器定义了 ExCtx 类,同时执行了 ExCtx 构建器,并将 ucl0 委派给了 ucl1。其中,在 ExCtx.java 文件中,第 24 行,基于装载约束规范,给出了完整的类型表达式如下:

$$<ExObj2, ucl0>^{ucl0} obj2 = <ExObj, ucl1>^{ucl0} value.ivar$$

既然必须保证 ucl0 和 ucl1 类装载器实例装载 ExObj2 类型的一致性,因此生成了装载约束 $Exobj2^{ucl0} = Exobj2^{ucl1}$ 。另外,由于在设置委派关系前,使用 ucl0 和 ucl1 装载了 ExObj2,从而破坏了约束,进而在运行过程中抛出了 LinkageError 异常。使用如下命令行运行实例:

```

[nr@toki examples]$ ant -Dchap=chap2 -Dex=0e run-example
Buildfile: build.xml
...
[java] java.lang.LinkageError: loader constraints violated when linking org/jboss/chap2/ex0/
ExObj2 class [java] at org.jboss.chap2.ex0.ExCtx.<init>(ExCtx.java:24)

```

```
[java] at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
[java] at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructor
AccessorImpl.java:39) [java] at sun.reflect.DelegatingConstructorAccessorImpl.newInstance
(DelegatingConstructorAccessorImpl.java:27) [java] at java.lang.reflect.Constructor.
newInstance(Constructor.java:274)
[java] at java.lang.Class.newInstance0(Class.java:308)
[java] at java.lang.Class.newInstance(Class.java:261)
[java] at org.jboss.chap2.ex0.ExLE.main(ExLE.java:53)
```

正如预期的一样，在 ExCtx 构建器中，第 24 行的代码破坏了装载约束，从而抛出了 LinkageError 异常。

4. 调试类装载问题

为获得类装载的位置信息，本文引入调试类装载问题。本书附带的例子：org.jboss.util.Debug 类（见列表 2-8），就是这样一个非常实用的工具。

列表 2-8 获得类的调试信息

```
Class clazz = ...;
StringBuffer results = new StringBuffer();
ClassLoader cl = clazz.getClassLoader();

results.append("\n"+clazz.getName()+"("+Integer.toHexString(clazz.hashCode())+"").Class
    Loader="+cl);
ClassLoader parent = cl;
while( parent != null )
{
    results.append("\n.."+parent);
    URL[] urls = getClassLoaderURLs(parent);
    int length = urls != null ? urls.length : 0;
    for(int u = 0; u < length; u++)
    {
        results.append("\n...."+urls[u]);
    }
    if( showParentClassLoaders == false )
        break;
    if( parent != null )
        parent = parent.getParent();
}

CodeSource clazzCS = clazz.getProtectionDomain().getCodeSource();
if( clazzCS != null )
    results.append("\n++++CodeSource: "+clazzCS);
else
    results.append("\n++++Null CodeSource");
```

其中，粗体的内容项很重要。首先，借助于 getClassLoader 方法能够获得各个 Class 对象的定义类装载器。这实际上定义了 Class 类型的范围，正如前面有关

ClassCastException、IllegalAccessException 及 LinkageError 的内容介绍的。通过类装载器，能够浏览到组成双亲委派链的 ClassLoader 的层次图。如果某 ClassLoader 为 URLClassLoader，则开发者还能看到用于类和资源装载的 URL。

ClassLoader 并不能通过 Class 的定义获得被装载 Class 的存放位置，然而通过 java.security.ProtectionDomain 和 java.security.CodeSource 却可以。其中，CodeSource 能够给出类的原始 URL 位置。但并不是每个 Class 都有 CodeSource，比如如果类是通过引导类装载器装载的，则 CodeSource 为空。总而言之，JVM 实现中的 java.* 和 javax.* 包都属于这种情形。

进一步而言，查看装载到 JBoss 服务器中的类细节可能更为有用。开发者能够通过使用 Log4j 配置文件（见列表 2-9 给出的配置片段）配置 JBoss 类装载层的日志输出详细程度。

列表 2-9 为实现详细的类装载日志服务的 log4j.xml 配置实例

```
<appender name="UCL" class="org.apache.log4j.FileAppender">
  <param name="File" value="${jboss.server.home.dir}/log/ucl.log"/>
  <param name="Append" value="false"/>
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%r,%c{1},%t] %m%n"/>
  </layout>
</appender>
<category name="org.jboss.mx.loading" additivity="false">
  <priority value="TRACE" class="org.jboss.logging.XLevel"/>
  <appender-ref ref="UCL"/>
</category>
```

其中，该配置将 org.jboss.mx.loading 包的类输出信息存储到服务器配置的 log 目录的 ucl.log 文件中。虽然查看类装载代码并没有实际意义，但是它对于提交 bug 报告或有关的类装载问题起到很重要的作用。如果开发者遇到可能是 bug 的类装载问题，请提交给 JBoss 项目（位于 SourceForge），并附上该.log 文件。如果.log 文件太大，请压缩后发到 scott.stark@jboss.org 信箱。

5. 深入 JBoss 类装载架构

到目前为止，本书已经阐述了 Java 定义的类型系统中类装载器的作用，接下来探讨 JBoss 3.x 的类装载架构。图 2-3 描述了类装载架构核心中的基本组件。

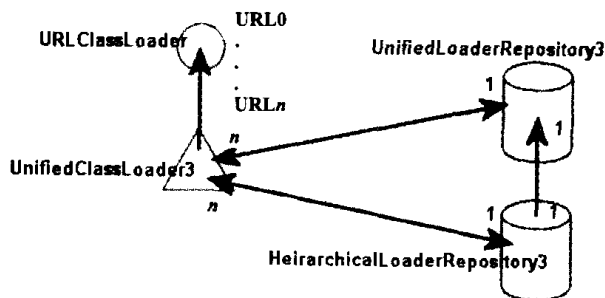


图 2-3 JBoss 3.x 核心类装载组件

中心组件是 `org.jboss.mx.loading.UnifiedClassLoader3`（UCL）类装载器。它扩展了标准的 `java.net.URLClassLoader`，其覆盖了标准的双亲委派模型以使用类和资源的共享库。其中，共享库为 `org.jboss.mx.loading.UnifiedLoaderRepository3`。另外，各个 UCL 只和单个 `UnifiedLoaderRepository3` 关联，但是通常情况下 `UnifiedLoaderRepository3` 具有多个 UCL。从 JBoss 3.0.5RC1 开始，UCL 可能和多个 URL 关联以实现类和资源的装载。在这之前，UCL 总是和单个的 URL 关联，因此由于存在包问题而易于出现 `IllegalAccessException` 和 `LinkageError` 异常。正如上节讨论的情形，即当存在多个类装载器时，这些错误是如何触发的一样。从 JBoss 3.0.5RC1 开始，部署器使用顶层部署 UCL 作为共享的类装载器，并将所有的部署存档都提交给该类装载器。“2.4.2 JBoss MBean 服务”有这方面的进一步阐述。

当触发 UCL 装载类时，它首先去库缓存区（repository cache）寻找是否已经装载了它。仅当在该库中找不到该类时，UCL 才会将它装载到库中。在默认情况下，仅存在单个的 `UnifiedLoaderRepository3`，它供所有的 UCL 实例共享使用，其意味着所有的 UCL 形成了单平面型（flat）的类装载器命名空间。当调用方法 `UnifiedClassLoader3.loadClass`（String, boolean）时，完整的执行步骤如下：

步骤

（1）检查与 `UnifiedClassLoader3` 关联的 `UnifiedLoaderRepository3` 类缓存区，寻找是否存在目标类缓存。如果找到，则将该缓存返回。

（2）否则，要求 `UnifiedClassLoader3` 去装载该类（如果它能够装载）。其中，必然会调用其双亲 `URLClassLoader.loadClass`（String, boolean）方法，从而能够判断该类是否处于该类装载器关联的 URL 中，或对其双亲类装载器可见。一旦寻找到，该类将被放置到类缓存区并返回给调用者。

（3）如果还是无法寻找到，则查询库以获得所有满足条件的 UCL，即那些能够基于库包名到 UCL 映射方式提供类的 UCL。当往库中添加 UCL 时，将创建与该 UCL 关联的 URL 中的包名之间的关联，从包名到 UCL 的映射也将再次更新。这使得能够快速定位哪个 UCL 能够装载该类。从而根据 UCL 添加到库的顺序，查找各个 UCL。一旦发现某个 UCL 能够装载该类，则将结果返回，否则将抛出 `java.lang.ClassNotFoundException` 异常。

（1）浏览装载库中的类

另外，能提供类信息的实用方式是 `UnifiedLoaderRepository3` 本身。它是 MBean 服务，包含了展示类和包信息的操作。通过标准的 JMX 名“`JMImplementation:name=Default,service=LoaderRepository`”，能够定位到默认库。同时，借助于 JMX 控制台能够访问到它，即通过“`http://localhost:8080/jmx-console/HtmlAdaptor?action=inspectMBean&name=JMImplementation%3AService%3DLoaderRepository%2Cname%3DDefault`”能够浏览到 JMX 控制台视图（见图 2-4）。下面的内容，即“2.3 连接到 JMX 服务器”一节将讨论 JMX 控制台。

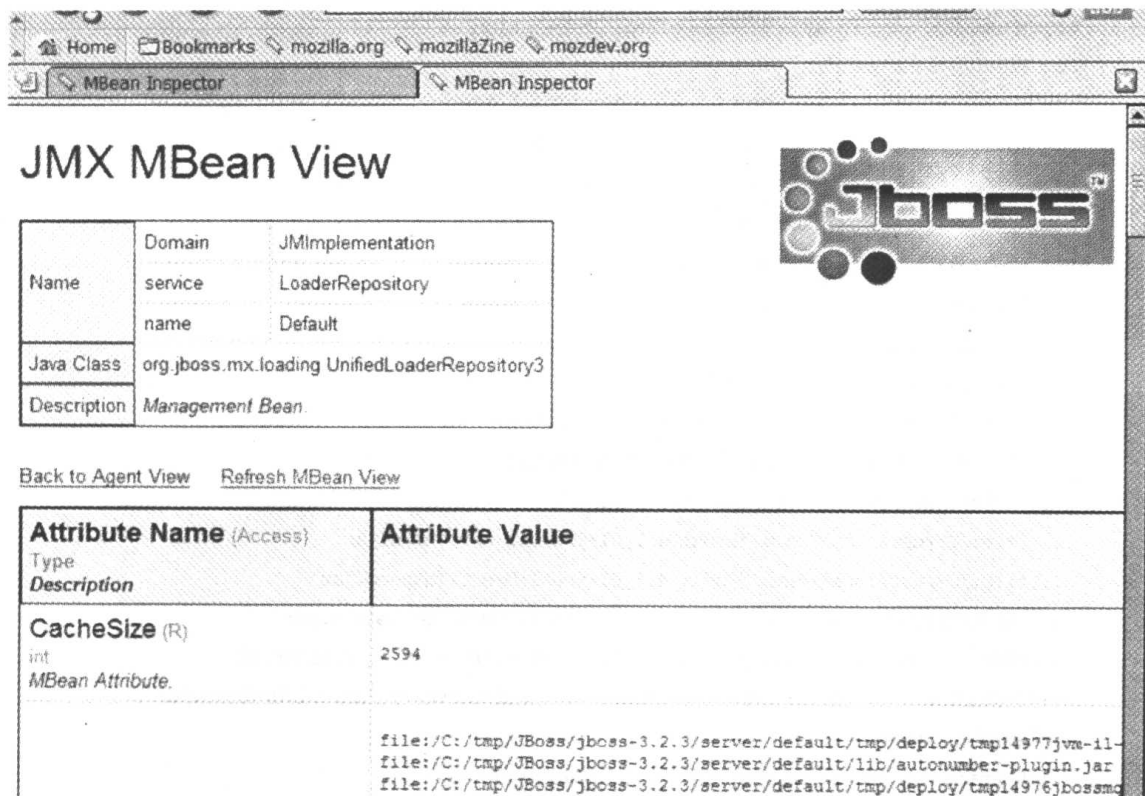


图 2-4 JMX 控制台中默认类 LoaderRepository MBean 视图

图 2-4 提供了两个实用的操作：`getPackageClassLoaders(String)`和 `displayClassInfo(String)`。其中，`getPackageClassLoaders` 操作返回类装载器集合，这些装载器建立了到指定包名中包含类或资源的映射。包名最后必须包含一个“.”。如果敲入包名“org.jboss.ejb.”，则结果如下：

```
[org.jboss.mx.loading.UnifiedClassLoader3@166a22b{ url=file:/C:/tmp/JBoss/jboss-3.0.5RC2/server/default/tmp/deploy/server/default/conf/jboss-service.xml/1.jbossservice.xml ,addedOrder=2}]
```

这是返回集合的字符串表示。它表明存在 `UnifiedClassLoader3` 实例，其主要 URL 指向“default/conf/jboss-service.xml”描述符。“addedOrder=2”表明它是第二个添加到库的类装载器。它持有服务器配置 lib 目录（如 server/default/lib）下的所有 jar 文件。如果敲入包名，则结果如下：

```
[org.jboss.mx.loading.UnifiedClassLoader3@47393f{ url=file:/C:/tmp/JBoss/jboss-3.0.5RC2/server/default/tmp/deploy/server/default/conf/jboss-service.xml/1.jbossservice.xml ,addedOrder=2},org.jboss.mx.loading.UnifiedClassLoader3@156e5ed{url=file:/C:/tmp/JBoss/jboss-3.0.5RC2/server/default/deploy/jmx-rmi-adaptor.sar/,addedOrder=6}]
```

这次存在两个 `UnifiedClassLoader3` 实例，一个为 default/conf/jboss-service.xml，另一个为 default/deploy/jmx-rmi-adaptor.sar。

通过传递类的全限定名给 `displayClassInfo` 的操作，开发者能够浏览特定类信息。比如，如果敲入上述实例包中的“org.jboss.jmx.adaptor.rmi.RMIAdaptorImpl”，将显示出如下结果：


```

org.jboss.jmx.adaptor.rmi.RMIAdaptorImpl Information
Repository cache version:
org.jboss.jmx.adaptor.rmi.RMIAdaptorImpl(11bd9c9).ClassLoader=org.jboss.mx.loading.Unified
ClassLoader3@166a22b{ url=file:/C:/tmp/JBoss/jboss-3.0.5RC2/server/default/tmp/deploy/server/
default/conf/jboss-service.xml/1.jboss-service.xml ,addedOrder=2}
..org.jboss.mx.loading.UnifiedClassLoader3@166a22b{ url=file:/C:/tmp/JBoss/jboss-3.0.5RC2/
server/default/tmp/deploy/server/default/conf/jboss-service.xml/1.jbossservice.xml ,addedOrder=2}
..org.jboss.system.server.NoAnnotationURLClassLoader@1bc4459
..sun.misc.Launcher$AppClassLoader@12f6684
....file:/C:/tmp/JBoss/jboss-3.0.5RC2/bin/
....file:/C:/usr/local/Java/j2sdk1.4.1_01/lib/tools.jar
....file:/C:/tmp/JBoss/jboss-3.0.5RC2/bin/run.jar
..sun.misc.Launcher$ExtClassLoader@f38798
....file:/C:/usr/local/Java/j2sdk1.4.1_01/jre/lib/ext/dnsns.jar
....file:/C:/usr/local/Java/j2sdk1.4.1_01/jre/lib/ext/ldapsec.jar
....file:/C:/usr/local/Java/j2sdk1.4.1_01/jre/lib/ext/localedata.jar
....file:/C:/usr/local/Java/j2sdk1.4.1_01/jre/lib/ext/sunjce_provider.jar
++++CodeSource: {file:/C:/tmp/JBoss/jboss-3.0.5RC2/server/default/lib/jboss.jar }
Implemented Interfaces:
++interface org.jboss.jmx.adaptor.rmi.RMIAdaptor(98f192)
++++ClassLoader: org.jboss.mx.loading.UnifiedClassLoader3@e31e33{ url=file:/C:/tmp/
JBoss/jboss-3.0.5RC2/server/default/deploy/jmx-rmi-adaptor.sar/ ,addedOrder=6}
++++CodeSource: {file:/C:/tmp/JBoss/jboss-3.0.5RC2/server/default/deploy/jmx-rmiadaptor.
sar/ )

### Instance0 found in UCL: org.jboss.mx.loading.UnifiedClassLoader3@166a22b{
url=file:/C:/tmp/JBoss/jboss-3.0.5RC2/server/default/tmp/deploy/server/default/conf/
jboss-service.xml/1.jboss-service.xml ,addedOrder=2}

```

其内容为装载库中特定 Class 实例信息的详细清单（前提是它已经被装载），后面的内容给出了存在该 Class 文件的类装载器。如果某类和多个类装载器关联，则存在类装载错误隐患。

（2）范围类

如果需要部署某应用的多个版本，JBoss 3.x 默认类装载模型则要求将各个应用部署在单独的 JBoss 服务器中。为获得安全性和资源监控的更多控制，往往需要这样做，但管理多个服务器实例并不是一件容易的事情。现存的一种替换机制是，使用基于范围（scoping）部署的方式来实现某应用多版本的部署。

对于基于范围部署的情形，各个部署创建 `HeirarchicalLoaderRepository3` 形式的、各自的类装载库。其中，JBoss 服务器在将包含在 EAR 中的部署单元实例委派给默认 `UnifiedLoaderRepository3` 前，`UnifiedClassLoader3` 会首先将其装载到自己的 `HeirarchicalLoaderRepository3` 中。为了实现 EAR 特定装载库，需要创建列表 2-10 所示的内容，并将其存放到 `META-INF/jboss-app.xml` 配置描述符中。

列表 2-10 为实现 EAR 级范围类的 jboss-app.xml 实例

```
<jboss-app>
  <loader-repository>some.dot.com:loader=webtest.ear</loader-repository>
</jboss-app>
```

其中, loader-repository 元素的取值是 JMX ObjectName, 从而为 EAR 创建自身的库。该值必须是惟一有效的 JMX ObjectName, 但实际上不是很重要。

(3) 完整的类装载模型

本书上面的内容在讨论核心类装载组件时, 仔细介绍了自定义的 UnifiedClassLoader3 和 UnifiedLoaderRepository3 类, 从而形成了共享类装载空间。完整的类装载描述必须包括 UnifiedClassLoader3 的双亲类装载器及其他用于范围和特殊用途类装载目的类装载器。图 2-5 给出了用于包含 EJB 和 WAR 的 EAR 部署的类层次纲要。

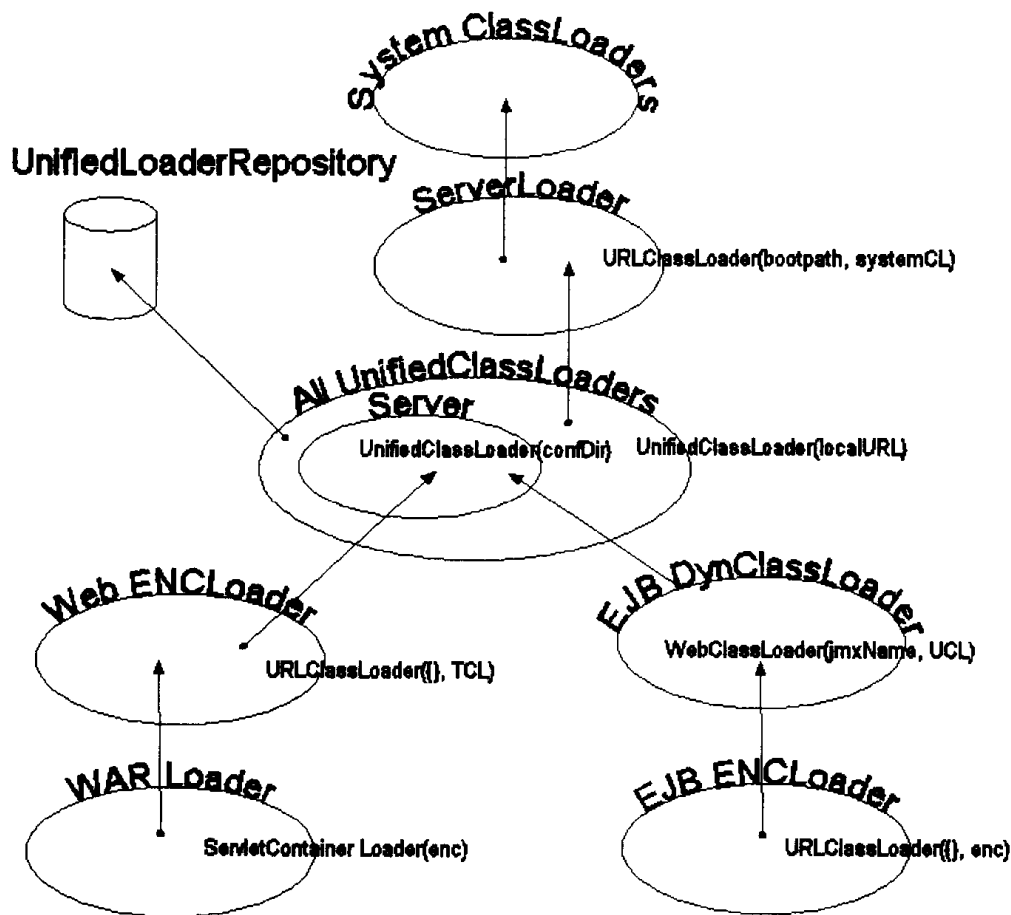


图 2-5 完整的类装载视图

图 2-5 相关内容的详细解释如下。

- 系统类装载器: 系统类装载器节点指 JVM 主线程, 或者运行嵌入式 JBoss 服务器的应用程序线程的线程上下文类装载器 (Thread Context Class Loader, TCL)。
- **ServerLoader:** ServerLoader 节点是指将类装载任务委派给系统类装载器的 URLClasserLoader。它包含如下一些引导 URL:

- 借助于 `jboss.boot.library.list` 系统属性而指定的所有 URL。目前，存在相对于由 `jboss.lib.url` 属性定义的 `libraryURL` 的路径约定。如果没有指定 `jboss.lib.url` 属性，则默认情况下为 `jboss.home.url + /lib/`。如果没有指定 `jboss.boot.library` 属性，则默认情况下指 `jaxp.jar`、`log4j-boot.jar`、`jboss-common.jar` 及 `jboss-system.jar`。
- JAXP jar 的具体取值依赖于主入口的 `-j` 选项。它或者是 `crimson.jar`，或者是 `xerces.jar`。在默认情况下为 `crimson.jar`。
- JBoss JMX jar (`jboss-jmx.jar`) 和 GNU regex jar (`gnu-regexp.jar`)。
- Oswego 的并行 jar, `concurrent.jar`。
- 借助于 `-L` 命令行选项指定的任何 jar 库文件。
- 借助于 `-C` 命令行选项指定的任何其他 jar 文件或目录。
- **Server**：Server 节点表示由 `org.jboss.system.server.Server` 接口实现创建的 `UnifiedClassLoader3` 集合。默认的接口实现除了为服务器 `conf` 目录创建 UCL 外，也为补丁目录 (`patchDir`) 入口创建 UCL。其中，创建的最后一个 UCL 作为 JBoss 主线程上下文类装载器。将来，JBoss 会将这些 UCL 合并到单个 UCL 中。现在，各个 UCL 支持多个 URL。
- **所有 UnifiedClassLoader3**：UnifiedClassLoader3 节点表示由部署器创建的 UCL。其中，凡是部署扫描器能够浏览到的内容，比如 EAR、JAR、WAR、SAR、目录、manifest 文件中引用的 jar 文件及包含的嵌入式部署单元，都被包含到这里的 UCL 中。由于形成的名字空间为平面类型，因此不会存在不同部署 jar 文件中类的多个实例。如果存在多个实例，则仅仅会使用到初次装载的类实例，但结果可能会和预期的存在差别。本文在“范围类”一节讨论了基于 EAR 部署单元的范围可见性机制。如果在特定的 JBoss 服务器部署某类的多个版本，则需要使用这种机制。
- **EJB DynClassLoader**：EJB DynClassLoader 节点继承于 `URLClassLoader` 中，它借助于简单的 HTTP WebService 提供 RMI 动态类装载。它指定了空的 `URL[]`，并将它委派给 `TCL`，将其作为双亲类装载器。如果配置的 HTTP WebService 允许装载系统级类，则借助于 HTTP 能够定位到 `UnifiedLoaderRepository3` 及系统 `classpath` 中所有的类。
- **EJB ENCLoader**：EJB ENCLoader 节点为 `URLClassLoader` 类型。其存在的目的只是为 EJB 部署单元的 `java:comp JNDI` 上下文提供惟一的上下文。它指定了空的 `URL[]`，并将它委派给 `EJB DynClassLoader`，将其作为双亲类装载器。
- **Web ENCLoader**：Web ENCLoader 节点为 `URLClassLoader` 类型。其存在的目的只是为 Web 部署单元的 `java:comp JNDI` 上下文提供惟一的上下文。它指定了空的 `URL[]`，并将它委派给 `TCL`，将其作为双亲类装载器。
- **WAR 装载器**：WAR 装载器是具体 Servlet 容器提供的类装载器。该类装载器委派给 `Web ENCLoader`，并将其作为双亲类装载器。其默认行为是，首先从双亲类装载器装载类，然后再去 `WEB-INF/{classes, lib}` 目录。如果开发者使用 Servlet 2.3 类装载模型，则首先从 WAR 的 `WEB-INF/{classes, lib}` 目录装载类，然后再

去其双亲类装载器。

JBoss 3.x 的类装载架构存在很多优缺点。其优点如下：

- 为了能够在不同部署单元实现类访问，不用再去不同部署单元中放置重复类。
- JBoss 后续版本可能会对库做新的划分，比如划分为域 (domain)、依赖及冲突检测，等等。

与此同时，相应的缺点如下：

- 为避免重复类，可能需要对现有的部署程序重新打包。位于装载库中的重复类依据类装载方式的不同，可能导致 `ClassCastException` 和 `LinkageError` 异常。从 JBoss 3.0.5RC1 发布版开始，这种情况一般不会发生，但还是可能发生。
- 需要隔离不同 EAR 中相同类的不同版本的部署，并通过 `jboss-app.xml` 配置描述符定义惟一的 `HeirarchicalLoaderRepository3`。

2.2.3 JBoss XMBean

XMBean 是 JMX `ModelMBean` 的 JBoss JMX 实现版本。XMBean 具有丰富的 `DynamicMBean` 元数据，而且并不需要通过乏味的编程以直接实现 `DynamicMBean` 接口所要求的开发内容。JBoss 的模型 `MBean` 实现允许通过 XML 描述符指定组件的管理接口。其中，XMBean 中的“X”即代表这里使用 XML 配置描述符的实现方式。XMBean 除了提供简单机制来描述 `DynamicMBean` 所要求的元数据外，它还能够实现属性持久化、缓存行为，甚至提供高级定制功能，比如 `MBean` 实现拦截器。图 2-6 展示了用于 XMBean 描述符的 `jboss_xmbean_1_0.dtd` 文件中的高层元素。该文件的详细展开细节请参考图 2-12。

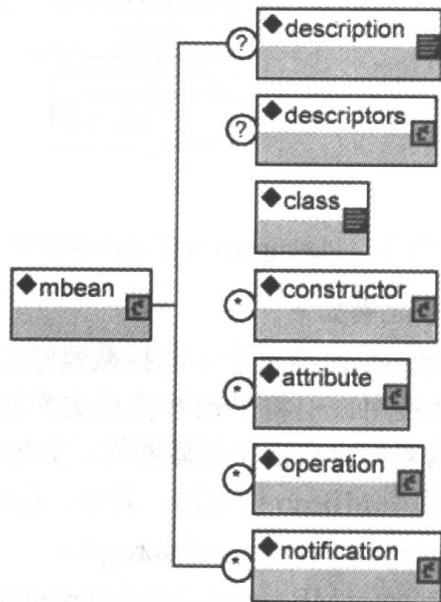


图 2-6 JBoss 1.0 XMBean DTD 概述 (jboss_xmbean_1_0.dtd)

图中，`mbean` 是文档的根元素，它包含了用于描述 `MBean` (`constructor`、`attribute`、`operation` 及 `notification`) 管理接口所要求的元素。同时，它还包括可选 `description` 元素以描述 `MBean` 的目的、可选 `descriptors` 元素以描述属性持久化策略、属性缓存等内容。

1. descriptors

descriptors 元素含有所含元素及子元素的所有描述符。无论是 JMX 规范建议的，还是 JBoss 使用的 descriptors 元素都存在预定义的元素和属性。其中，自定义 descriptors 存在普通 descriptor 元素，该元素具有 name 和 value 属性。图 2-7 展示了 descriptors 元素的内容模型。

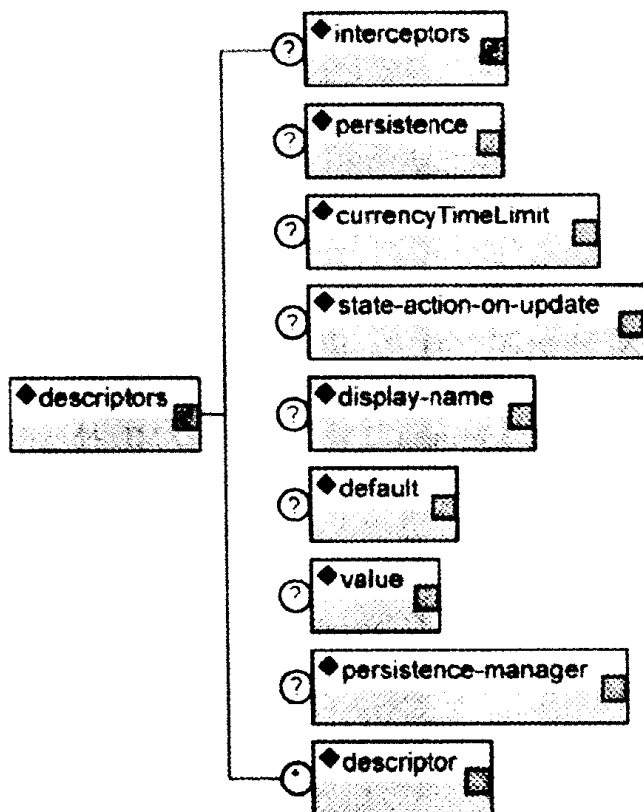


图 2-7 descriptors 元素的内容模型

descriptors 元素中的主要子元素如下。

- **interceptors:** interceptors 元素用于定制拦截器栈以替代默认栈。当前，这种做法只能在 MBean 级使用，但将来能够在自定义属性或操作级使用拦截器栈。interceptors 元素的内容指定自定义拦截器栈。如果没有指定 interceptors 元素，JBoss 将使用标准的模型 MBean 拦截器。其中，标准的拦截器有：

- org.jboss.mx.interceptor.PersistenceInterceptor
- org.jboss.mx.interceptor.MBeanAttributeInterceptor
- org.jboss.mx.interceptor.ObjectReferenceInterceptor

当指定自定义的拦截器栈时，通常都会包括标准的拦截器，除非开发者打算替代它们。

每个 interceptors 元素的取值需要给出拦截器实现的全限定类名。其中，该类必须实现 org.jboss.mx.interceptor.Interceptor 接口。它的构建器或者为无参，或接受 (javax.management.MBeanInfo, org.jboss.mx.server.MBeanInvoker) 参数对。

interceptors 元素可能存在若干属性。其中, 这些属性对应于拦截器类实现中以 **JavaBean** 风格实现的属性。对于各个指定的 **interceptor** 元素属性, JBoss 将查询拦截器类以寻找到匹配它的 **setter** 方法。通过使用与类型关联的 **java.beans.PropertyEditor** 能够将属性值转换为拦截器类属性的实际类型。如果不存在属性对应的 **setter** 方法或与之关联的 **PropertyEditor** 时, 在 **interceptors** 元素中指定该属性将触发错误出现。

- **persistence**: **persistence** 元素包含了 JMX 规范建议的 **persistPolicy**、**persistPeriod**、**persistLocation** 及 **persistName** 属性。其各自的含义如下:
 - **persistPolicy**: 该属性定义了何时持久化属性, 而且其取值范围如下:
 - ✓ **Never**: 表明属性从不持久化。
 - ✓ **OnUpdate**: 当属性值发生变化时, 持久它。
 - ✓ **OnTimer**: 基于 **persistPeriod** 指定的时间实现属性的持久化。
 - ✓ **NoMoreOftenThan**: 一旦属性值发生变化, 便持久它。但前提是, 持久的频率不能够超过 **persistPeriod** 指定的时间。
 - **persistPeriod**: 当 **persistPolicy** 属性值为 **OnTimer** 或 **NoMoreOftenThan** 时, **persistPeriod** 指定了属性的更新频率 (单位: 毫秒)。
 - **persistLocation**: **persistLocation** 属性指定持久源的位置, 其具体的表达方式取决于 JMX 持久化实现。目前, 如果使用 JBoss 默认持久化管理器, 属性将被序列化到某个指定目录。
 - **persistName**: 该属性与 **persistLocation** 一起使用, 从而进一步指定持久源的位置。比如, 如果 **persistLocation** 属性取值为目录, 则 **persistName** 将指定文件名, 以在该目录下存储属性。
- **currencyTimeLimit**: **currencyTimeLimit** 元素指定被缓存属性值 (单位: 秒) 的有效时限。如果 **currencyTimeLimit** 取值为 0, 则表明总是应该从 **MBean** 中获得属性值, 而不去缓存它。如果 **currencyTimeLimit** 取值为 -1, 则表明被缓存的属性值从不失效。
- **state-action-on-update**: **state-action-on-update** 元素给出当更新任一属性时 **MBean** 会采取的行动, 即 **MBean** 生命周期状态的改变。行动的内容由该属性值指定, 其具体取值范围为:
 - **keep-running**
 - **restart**
 - **reconfigure**
 - **reinstantiate**

然而, JBoss 当前并未使用该元素。

- **display-name**: **display-name** 元素描述某项内容, 以便于人们更好地理解其含义。
- **default**: 当未指定某属性时, **default** 元素给出其默认取值。有一点请注意, 即在启动 **MBean** 时, 该值并不会像 **jboss-service.xml** 中的元素内容取值而保留在 **MBean** 中。该属性值仅仅用于没有定义访问属性的入口方法及 **value** 元素的场景。

- **value:** value 元素指定管理属性的当前值。同 default 元素不同，在启动 MBean 时，它的取值能够通过定义的 setter 方法写入到 MBean 中。
- **persistence-manager:** persistence-manager 元素给出类名，作为持久化管理器。其中，该类名实现了 org.jboss.mx.persistence.PersistenceManager 接口。当前，JBoss 仅提供 org.jboss.mx.persistence.ObjectStreamPersistenceManager 持久化管理器，它使用 Java 序列化将 ModelMBeanInfo 内容存储到文件中。
- **descriptor:** descriptor 元素指定 JBoss 之外的值对。其 name 属性指明该 descriptor 元素类型，value 属性指明该 descriptor 元素值。descriptor 元素用于提供附加的管理元数据。

请注意，constructor、attribute、operation 及 notification 元素可能都含有 descriptors 元素，它指定了规范要求的 descriptors 内容和期望的描述符扩展设置。

2. 管理类

class 元素用于指定管理对象的全限定名，其管理接口通过 XMLElement 描述符给出。

3. 构建器

为创建管理对象实例，constructor 元素指明了可用的构建器。图 2-8 给出了 constructor 元素及其内容模型。

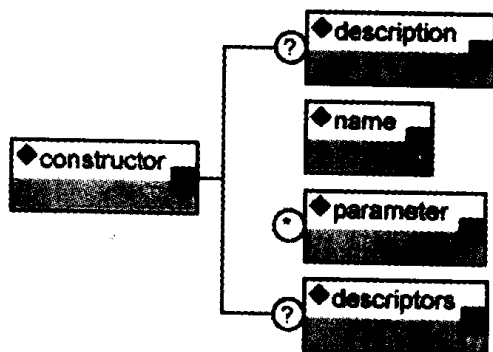


图 2-8 XMLElement constructor 元素及其内容模型

它包含的几个较重要子元素如下：

- **description:** 构建器的描述。
- **name:** 构建器的名字，其内容必须和实现类相同。
- **parameter:** 描述构建器参数。其中，参数值具备如下属性：
 - description: 参数的可选描述。
 - name: 要求的参数变量名。
 - type: 要求的参数类型全限定类名。
- **descriptors:** 与构建器元数据关联的任意 descriptors。

4. 属性

attribute 元素指定 MBean 暴露的管理属性。图 2-9 描述了 attribute 元素及其内容模型。

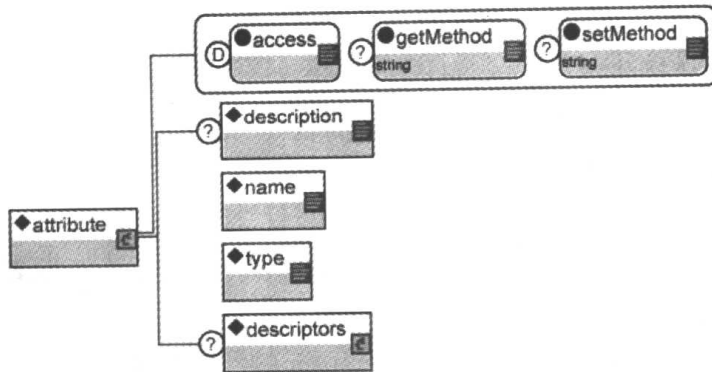


图 2-9 XMLElement attribute 元素及其内容模型

attribute 元素支持的属性如下:

- **access**: 它为可选元素, 用于定义属性的读/写模式。其具体取值范围如下:
 - read-only: 只读属性。
 - write-only: 只写属性。
 - read-write: 默认情况下, 该属性为可读、可写。
- **getMethod**: 该元素定义了方法名, 以读取命名方法名。如果需要从 MBean 实例获得该管理属性, 则必须指定该元素。
- **setMethod**: 该元素定义了方法名, 以写入命名方法名。如果需要从 MBean 实例获得该管理属性, 则必须指定该元素。

attribute 元素其余较重要的子元素如下:

- **description**: attribute 元素的描述。
- **name**: attribute 的名称, 供 MBeanServer.getAttribute()操作使用。
- **type**: 属性类型的全限定类名。
- **descriptors**: 其他任意 descriptors, 比如属性持久化、缓存、默认值, 等等。

5. 操作

XMLElement 需要借助于一个或多个 operation 元素暴露其管理操作。图 2-10 展示了 operation 元素及其内容模型。

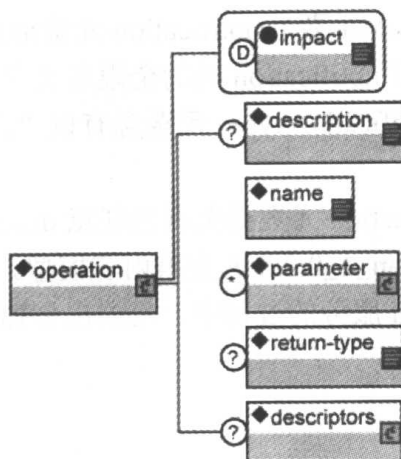


图 2-10 XMLElement operation 元素及其内容模型

其中, **impact** 属性定义对操作执行的影响, 其具体取值范围如下:

- **ACTION**: 该操作改变 MBean 组件的状态 (写操作)。
- **INFO**: 该操作不改变 MBean 组件的状态 (读操作)。
- **ACTION_INFO**: 类似于读、写操作。

operation 的子元素如下:

- **description**: **description** 元素为 **operation** 元素给出便于开发者理解的描述信息。
- **name**: **name** 元素给出 **operation** 名。
- **parameter**: **parameter** 元素给出 **operation** 的参数定义。
- **return-type**: **return-type** 元素定义操作返回类型的全限定类名。如果没有指定, 则为 **void** 类型。
- **descriptors**: 同 **operation** 元数据关联的任意 **descriptors**。

6. 通知

notification 元素描述 XMBean 提交的管理通知。图 2-11 给出了 **notification** 元素及其内容模型。

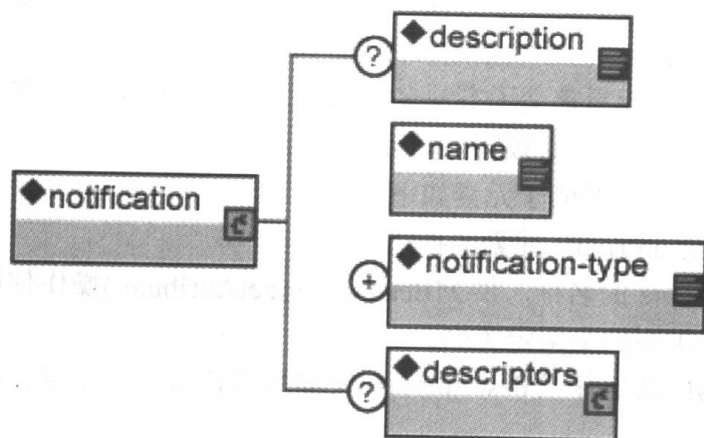


图 2-11 XMBean notification 元素及其内容模型

其子元素如下:

- **description**: **description** 元素为 **notification** 元素给出便于开发者理解的描述信息。
- **name**: **name** 元素含有 **notification** 类的全限定类名。
- **notification-type**: **notification-type** 元素含有以 “.” 隔开的字符串, 它用于描述通知类型。
- **descriptors**: 同 **notification** 元数据关联的任意 **descriptors**。

图 2-12 给出了 **jboss_xmbean_1_0.dtd** 的完整内容模型。在 2.4.3 中的 “2. XMBean 实例” 一节, 在讨论 JBoss MBean 服务的过程中, 将给出创建 XMBean 实例的完整过程。

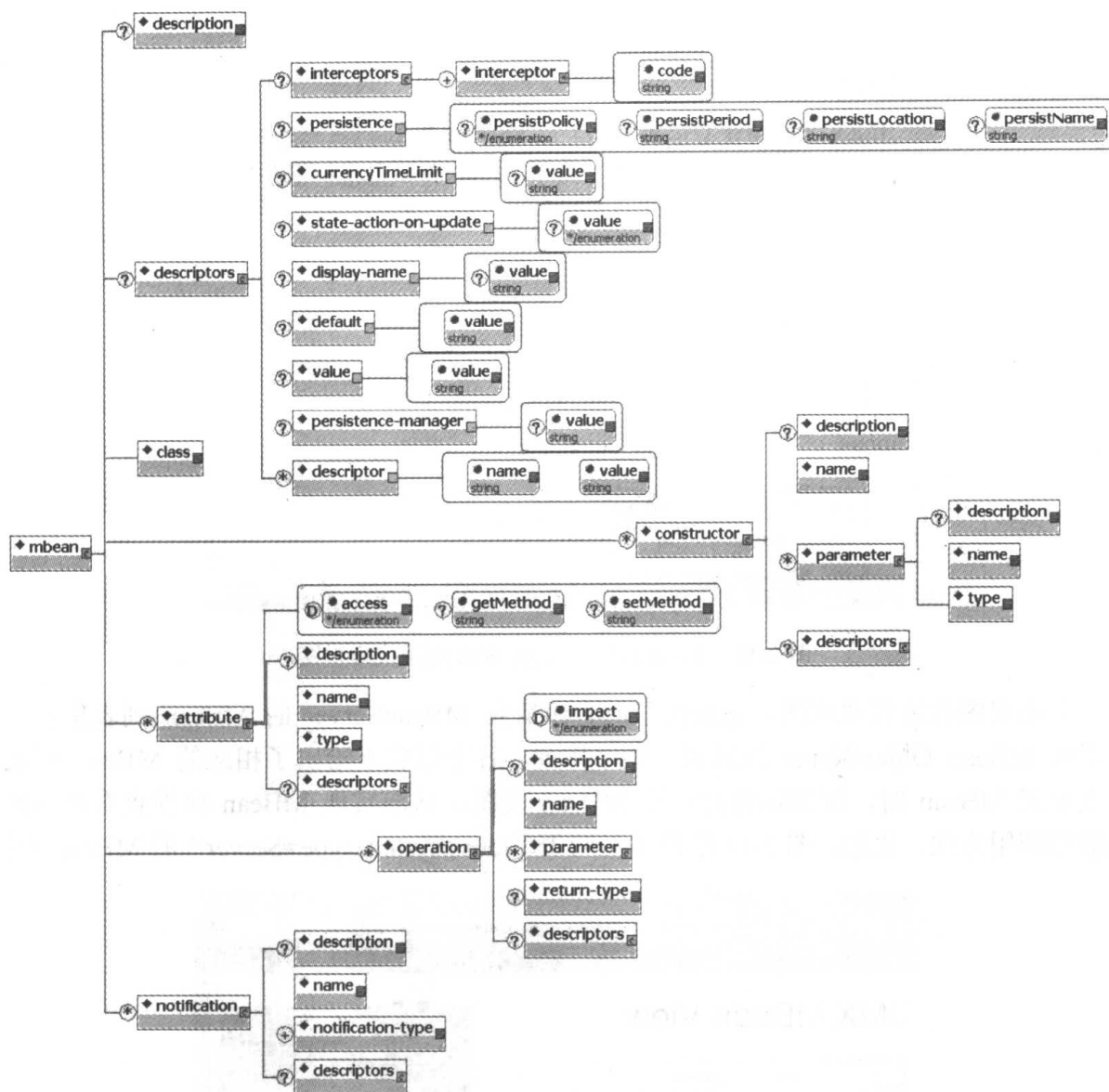


图 2-12 jboss_xmbean_1_0.dtd 的展开视图

2.3 连接到 JMX 服务器

为从运行 JBoss 服务器虚拟机外部实现对 JMX MBeanServer 的访问, JBoss 提供了相应的适配器。目前, JBoss 提供的适配器包括 HTML、RMI 接口及 EJB 等内容。

2.3.1 浏览服务器——JMX 控制台 Web 应用

从 JBoss 3.0.1 开始, JBoss 提供了其实现的 JMX HTML 适配器, 即允许通过标准 Web 浏览器查看 MBeanServer 中的 MBean。控制台 Web 应用的默认 URL 地址为 <http://localhost:8080/jmx-console/>。用户打开该 URL 后, 将出现类似于图 2-13 所示的结果。

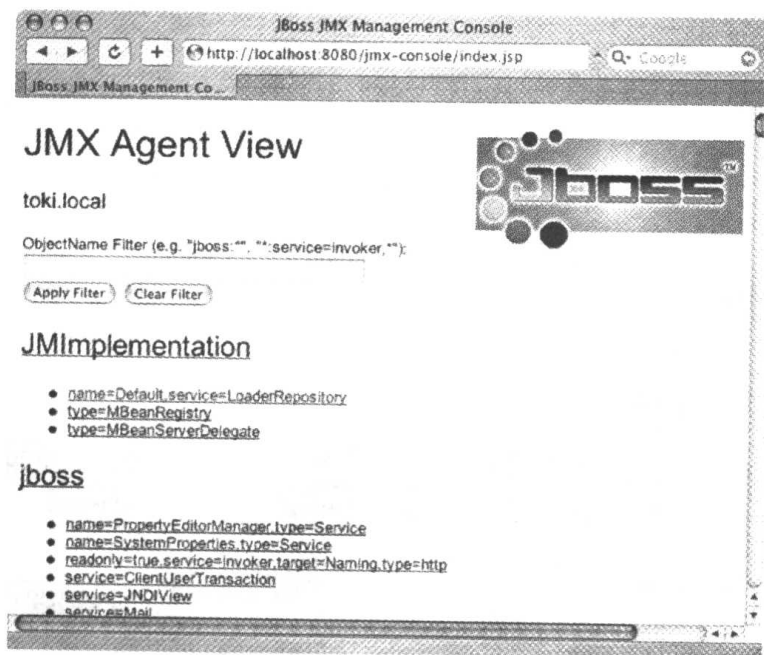


图 2-13 JBoss JMX 控制台 Web 应用代理视图

上述视图称为代理视图。它提供了所有注册到 MBeanServer 的 MBean 列表集合，并且根据 MBean ObjectName 的域划分顺序排列。各个域底下给出了相应的 MBean 列表。当选中某 MBean 时，浏览器将打开该 MBean 视图，从而实现 MBean 属性查看和编辑，并能够调用方法。比如，图 2-14 给出了对应于“jboss.system:type=Server”的 MBean 视图。

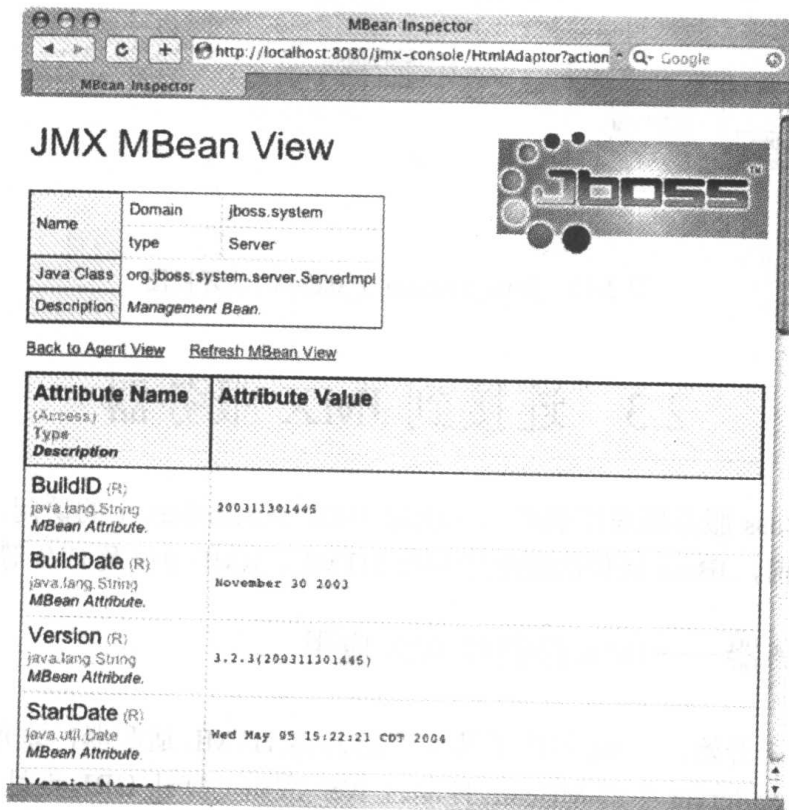


图 2-14 “jboss.system:type=Server” MBean 的视图

JMX 控制台 Web 应用的源代码位于 jboss-all/varia 模块的 src/main/org/jboss/jmx 目录。另外, 其 Web 页面位于 jboss-all/varia/src/resources/jmx 目录。该应用借助于 MBeanServer, 实现了基于 JSP 视图和 Servlet 的 MVC 模式实现。

保护 JMX 控制台

由于 JMX 控制台 Web 应用只是标准的 Servlet, 因此开发者能够使用基于安全性的标准 J2EE 角色来保护它。同时, 该控制台 Web 应用自从 JBoss 3.0.1 发布版开始就包括在 JBoss 中。部署的 jmx-console.war 并没有打包在 .war 中, 使得开发者能够快速编辑简单用户名和密码, 以获得安全性约束功能。如果开发者查看 server/default/deploy 目录中的 jmx-console.war, 将在 Web-INF 目录找到 web.xml 和 jboss-web.xml 描述符, 同时还能够在 Web-INF/classes 目录找到 roles.properties 和 users.properties 文件。具体操作如下:

```
[nr@toki jboss-3.2.3]$ ls server/default/deploy/jmx-console.war/WEB-INF/classes jboss-web.xml
web.xml
[nr@toki jboss-3.2.3]$ ls server/default/deploy/jmx-console.war/WEB-INF/classes/org roles.
properties users.properties
```

开发者在解开 web.xml 和 jboss-web.xml 描述符中安全性部分的注释后 (见列表 2-11), HTTP basic 认证便生效了, 即仅仅能够通过用户名 admin、密码 admin 实现 jmx-console 应用程序的约束访问。其中, 上述用户名是由 Web-INF/classes/users.properties 文件中的 admin=admin 行决定的。

列表 2-11 jmx-console.war 中解开安全性元素后的 web.xml 和 jboss-web.xml 描述符

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

    <!-- A security constraint that restricts access to the HTML JMX console
    to users with the role JBossAdmin. Edit the roles to what you want and
    uncomment the WEB-INF/jboss-web.xml/security-domain element to enable
    secured access to the HTML JMX console.
    -->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>HtmlAdaptor</web-resource-name>
            <description>An example security config that only allows users with the
            role JBossAdmin to access the HTML JMX console web application
            </description>
            <url-pattern>/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
```



```
<auth-constraint>
  <role-name>JBossAdmin</role-name>
</auth-constraint>
</security-constraint>

<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>JBoss JMX Console</realm-name>
</login-config>

<security-role>
  <role-name>JBossAdmin</role-name>
</security-role>
</web-app>

<jboss-web>
  <!-- Uncomment the security-domain to enable security. You will
       need to edit the htmldaptor login configuration to setup the
       login modules used to authentication users.
  -->
  <security-domain>java:/jaas/jmx-console</security-domain>
</jboss-web>
```

然后, 开发者需要保存相应的修改, 并启动服务器 (如果没有启动), 试试访问 jmx-console URL。最后, 开发者能看到类似于图 2-15 所示的对话框。

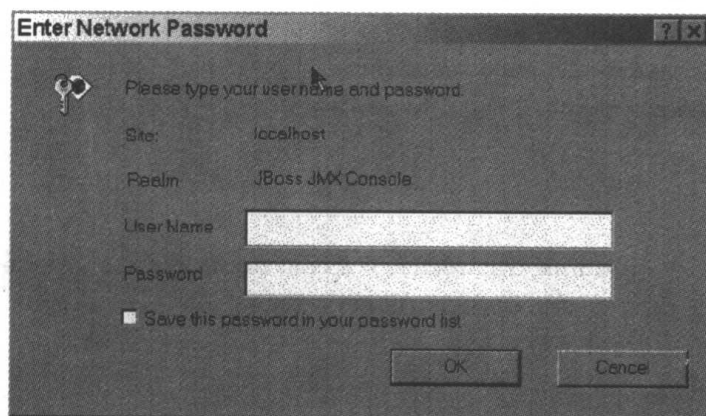


图 2-15 保存列表 2-11 后展示出 jmx-console basic HTTP 登录对话框

在通常情况下, 采用 .properties 文件来保护对 JMX 控制台应用的做法很不安全。本书第 8 章将详细阐述如何正确地配置 Web 应用的安全性设置。

2.3.2 使用 RMI 连接到 JMX

JBoss 提供了 RMI 接口, 以连接到 JMX MBeanServer。其接口为 (见列表 2-12): org.jboss.jmx.adaptor.rmi.RMIAdaptor。

列表 2-12 RMIAdaptor 接口

```
/*
 * JBoss, the OpenSource J2EE webOS
 *
 * Distributable under LGPL license.
 * See terms of license at gnu.org.
 */
package org.jboss.jmx.adaptor.rmi;

import javax.management.Attribute;
import javax.management.AttributeList;
import javax.management.ObjectName;
import javax.management.QueryExp;
import javax.management.ObjectInstance;
import javax.management.NotificationFilter;
import javax.management.NotificationListener;
import javax.management.MBeanInfo;

import javax.management.AttributeNotFoundException;
import javax.management.InstanceAlreadyExistsException;
import javax.management.InstanceNotFoundException;
import javax.management.IntrospectionException;
import javax.management.InvalidAttributeValueException;
import javax.management.ListenerNotFoundException;
import javax.management.MBeanException;
import javax.management.MBeanRegistrationException;
import javax.management.NotCompliantMBeanException;
import javax.management.OperationNotSupportedException;
import javax.management.ReflectionException;

public interface RMIAdaptor
    extends java.rmi.Remote
{

    public ObjectInstance createMBean(String pClassName, ObjectName pName)
        throws ReflectionException,
            InstanceAlreadyExistsException,
            MBeanRegistrationException,
            MBeanException,
            NotCompliantMBeanException,
            RemoteException;

    public ObjectInstance createMBean(String pClassName, ObjectName pName,
        ObjectName pLoaderName)
        throws ReflectionException,
```

```
InstanceAlreadyExistsException,
MBeanRegistrationException,
MBeanException,
NotCompliantMBeanException,
InstanceNotFoundException,
RemoteException;

public ObjectInstance createMBean(String pClassName, ObjectName pName,
    Object[] pParams, String[] pSignature)
    throws ReflectionException,
    InstanceAlreadyExistsException,
    MBeanRegistrationException,
    MBeanException,
    NotCompliantMBeanException,
    RemoteException;

public ObjectInstance createMBean(String pClassName, ObjectName pName,
    ObjectName pLoaderName, Object[] pParams, String[] pSignature)
    throws ReflectionException,
    InstanceAlreadyExistsException,
    MBeanRegistrationException,
    MBeanException,
    NotCompliantMBeanException,
    InstanceNotFoundException,
    RemoteException;

public void unregisterMBean(ObjectName pName)
    throws InstanceNotFoundException,
    MBeanRegistrationException,
    RemoteException;

public ObjectInstance getObjectInstance(ObjectName pName)
    throws InstanceNotFoundException,
    RemoteException;

public Set queryMBeans(ObjectName pName, QueryExp pQuery)
    throws RemoteException;

public Set queryNames(ObjectName pName, QueryExp pQuery)
    throws RemoteException;

public boolean isRegistered(ObjectName pName)
    throws RemoteException;

public boolean isInstanceOf(ObjectName pName, String pClassName)
    throws InstanceNotFoundException,
    RemoteException;
```

```
public Integer getMBeanCount()
    throws RemoteException;

public Object getAttribute(ObjectName pName, String pAttribute)
    throws MBeanException,
    AttributeNotFoundException,
    InstanceNotFoundException,
    ReflectionException,
    RemoteException;

public AttributeList getAttributes(ObjectName pName, String[] pAttributes)
    throws InstanceNotFoundException,
    ReflectionException,
    RemoteException;

public void setAttribute(ObjectName pName, Attribute pAttribute)
    throws InstanceNotFoundException,
    AttributeNotFoundException,
    InvalidAttributeValueException,
    MBeanException,
    ReflectionException,
    RemoteException;

public AttributeList setAttributes(ObjectName pName, AttributeList pAttributes)
    throws InstanceNotFoundException,
    ReflectionException,
    RemoteException;

public Object invoke(ObjectName pName, String pActionName,
    Object[] pParams, String[] pSignature)
    throws InstanceNotFoundException,
    MBeanException,
    ReflectionException,
    RemoteException;

public String getDefaultDomain()
    throws RemoteException;

public void addNotificationListener(ObjectName pName, ObjectName pListener,
    NotificationFilter pFilter, Object pHandback)
    throws InstanceNotFoundException,
    RemoteException;

public void removeNotificationListener(ObjectName pName, ObjectName pListener)
    throws InstanceNotFoundException,
```

```
ListenerNotFoundException,  
RemoteException;  
  
public MBeanInfo getMBeanInfo(ObjectName pName)  
    throws InstanceNotFoundException,  
        IntrospectionException,  
        ReflectionException,  
        RemoteException;  
}
```

其中，RMIAaptor 接口被 org.jboss.jmx.adaptor.rmi.RMIAaptorService MBean 服务绑定到 JNDI 中。尽管从 JBoss 3.2.2 发布版开始，已经将该服务从 deploy 目录删除，但是通过 docs/examples/jmx 目录还是能够找到该服务。同时，由于 JBoss 推荐使用通过 Invoker 适配器服务连接到 JMX MBeanServer，因此这种方法已经被 JBoss 丢弃。当然，考虑到与现有客户的兼容性，Invoker 适配器服务还是支持 RMIAaptor 接口，并将其绑定到默认位置“jmx/rmi/RMIAaptor”。另外，“2.7.1 分离式 Invoker 实例：MBeanServer Invoker 适配器服务”一节详细讨论了 Invoker 适配器服务。RMIAaptorService 仍然为远程客户提供接收 JMX 通知提供服务。由于 Invoker 适配器服务并不具备这项功能，因此如果需要，则必须将 jmx-rmi-adaptor.sar（位于光盘目录）替代 jmx-invoker-adaptor-server.sar 存档。

RMIAaptorService 以 jmx-rmi-adaptor.sar 的形式部署，而且支持下列属性。

- **JndiName**：绑定 RMIAaptor 接口的 JNDI 名字。默认的名字为 jmx/rmi/RMIAaptor。在 JBoss 3.0.4 之前，该 JNDI 名字硬编码（hard-coded）成“jmx:” + <server> + “:rmi”。其中，<server>取值可以通过 InetAddress.getLocalHost().getHostName()获得。考虑到后向兼容性，目前该绑定仍然存在，在后续版本中有可能被丢弃。
- **RMIObjectPort**：用于导出（exported）RMI 对象的服务器端监听端口号。在默认情况下，其取值为 0，即允许 JBoss 开发者选择匿名的可用端口。
- **ServerAddress**：与 RMIObjectPort 对应，用于导出 RMI 对象的服务器接口名或 IP 地址。在默认情况下为空值，即允许绑定到所有可用的接口。
- **BackLog**：在连接错误出现前，RMI 对象服务器 Socket 还未处理完成的客户连接请求。

列表 2-13 给出了某客户应用，它利用 RMIAaptor 接口实现 JNDIView MBean 中的 MBeanInfo 信息查询。该应用也调用了 MBean 的 list(boolean)方法，并将结果显示出来。

列表 2-13 使用 RMIAaptor 的 JMX 客户

```
package org.jboss.chap2.ex4;  
  
import javax.management.MBeanInfo;  
import javax.management.MBeanOperationInfo;  
import javax.management.MBeanParameterInfo;  
import javax.management.ObjectName;  
import javax.naming.InitialContext;
```



```
import org.jboss.jmx.adaptor.rmi.RMIAdaptor;

/** A client that demonstrates how to connect to the JMX server using the RMI
    adaptor.

    @author Scott.Stark@jboss.org
    @version $Revision: 1.2 $
    */
public class JMXBrowser
{

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws Exception
    {
        InitialContext ic = new InitialContext();
        RMIAdaptor server = (RMIAdaptor) ic.lookup("jmx/rmi/RMIAdaptor");

        // Get the MBeanInfo for the JNDIView MBean
        ObjectName name = new ObjectName("jboss:service=JNDIView");
        MBeanInfo info = server.getMBeanInfo(name);
        System.out.println("JNDIView Class: "+info.getClassName());
        MBeanOperationInfo[] opInfo = info.getOperations();
        System.out.println("JNDIView Operations: ");
        for(int o = 0; o < opInfo.length; o ++)
        {
            MBeanOperationInfo op = opInfo[o];
            String returnType = op.getReturnType();
            String opName = op.getName();
            System.out.print(" + "+returnType+" "+opName+"(");
            MBeanParameterInfo[] params = op.getSignature();
            for(int p = 0; p < params.length; p ++)
            {
                MBeanParameterInfo paramInfo = params[p];
                String pname = paramInfo.getName();
                String type = paramInfo.getType();
                if( pname.equals(type) )
                    System.out.print(type);
                else
                    System.out.print(type+" "+pname);
                if( p < params.length-1 )
                    System.out.print(',');
            }
            System.out.println(")");
        }
    }
}
```

```
}

// Invoke the list(boolean) op
String[] sig = {"boolean"};
Object[] opArgs = {Boolean.TRUE};
Object result = server.invoke(name, "list", opArgs, sig);
System.out.println("JNDIView.list(true) output:\n"+result);
}
}
```

为测试 RMIAdaptor 客户应用，运行如下命令：

```
[orb@toki examples]$ ant -Dchap=chap2 -Dex=4 run-example
Buildfile: build.xml
```

validate:

```
[java] ImplementationTitle: JBoss [WonderLand]
[java] ImplementationVendor: JBoss.org
[java] ImplementationVersion: 3.2.3 (build: CVSTag=JBoss_3_2_3 date=200311301445)
[java] SpecificationTitle: JBoss
[java] SpecificationVendor: JBoss (http://www.jboss.org/)
[java] SpecificationVersion: 3.2.3
[java] JBoss version is: 3.2.3
```

fail_if_not_valid:

init:

```
[echo] Using jboss.dist=/Users/orb/java/jboss-3.2.3
```

compile:

```
[javac] Compiling 3 source files to /Users/orb/Desktop/jboss/AdminDevel 2/examples/output/classes
```

run-example:

run-example4:

```
[java] JNDIView Class: org.jboss.mx.modelmbean.XMBean
[java] JNDIView Operations:
[java] + java.lang.String list(boolean jboss:service=JNDIView)
[java] + java.lang.String listXML()
[java] + void create()
[java] + void start()
[java] + void stop()
[java] + void destroy()
[java] + java.lang.String getName()
[java] + int getState()
[java] + java.lang.String getStateString()
[java] JNDIView.list(true) output:
```

```

[java] <h1>Ejb Module: ClusteredHttpSessionEB.jar</h1>
[java] <h2>java:comp namespace of the ClusteredHTTPSession bean:</h2>
[java] <pre>
[java] +- env (class: org.jnp.interfaces.NamingContext)
[java] </pre>
[java] <h1>java: Namespace</h1>
[java] <pre>
[java] +- jaas (class: javax.naming.Context)
[java] | +- jbossmq-httpil (class: org.jboss.security.plugins.SecurityDomainContext)
[java] | +- other (class: org.jboss.security.plugins.SecurityDomainContext)
[java] | +- JmsXARealm (class: org.jboss.security.plugins.SecurityDomainContext)
[java] | +- jbosstest-web (class: org.jboss.security.plugins.SecurityDomainContext)
[java] | +- http-invoker (class: org.jboss.security.plugins.SecurityDomainContext)
[java] | +- jbossmq (class: org.jboss.security.plugins.SecurityDomainContext)
[java] | +- secure-jndi (class: org.jboss.security.plugins.SecurityDomainContext)
[java] | +- HsqlDbRealm (class: org.jboss.security.plugins.SecurityDomainContext)
[java] +- TransactionPropagationContextImporter (class: org.jboss.tm.TransactionPropagation
ContextImporter)
[java] +- JmsXA (class: org.jboss.resource.adapter.jms.JmsConnectionFactoryImpl)
[java] +- JBossCorbaNaming (class: org.omg.CosNaming.NamingContextExt)
[java] +- DefaultDS (class: org.jboss.resource.adapter.jdbc.WrapperDataSource)
[java] +- StdJMSPool (class: org.jboss.jms.asf.StdServerSessionPoolFactory)
[java] +- TransactionManager (class: org.jboss.tm.TxManager)
[java] +- JBossCorbaPOA (class: org.omg.PortableServer.POA)
[java] +- TransactionPropagationContextExporter (class: org.jboss.tm.TransactionPropagation
ContextFactory) [java] +- ConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
[java] +- DefaultJMSProvider (class: org.jboss.jms.jndi.JBossMQProvider)
[java] +- XAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
[java] +- JBossCorbaInterfaceRepositoryPOA (class: org.omg.PortableServer.POA)
[java] +- Mail (class: javax.mail.Session)
[java] +- JBossCorbaORB (class: org.omg.CORBA.ORB)
[java] +- cmrTransactionTest (class: org.jnp.interfaces.NamingContext)
[java] +- timedCacheFactory (class: javax.naming.Context)
[java] Failed to lookup: timedCacheFactory, errmsg=null
[java] +- SecurityProxyFactory (class: org.jboss.security.SubjectSecurityProxyFactory)
[java] +- comp (class: javax.naming.Context)
[java] </pre>
[java] <h1>Global JNDI Namespace</h1>
[java] <pre>
[java] +- OIL2ConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
[java] +- hello (class: org.jnp.interfaces.NamingContext)
[java] +- HAPartition (class: org.jnp.interfaces.NamingContext)
[java] | +- DefaultPartition (class: org.jboss.ha.framework.server.HAPartitionImpl)
[java] +- helloworld (class: org.jnp.interfaces.NamingContext)
[java] +- jbosstest (class: org.jnp.interfaces.NamingContext)
[java] | +- ejbs (class: org.jnp.interfaces.NamingContext)

```

```
[java] | | +- local (class: org.jnp.interfaces.NamingContext)
[java] +- XAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
[java] +- queue (class: org.jnp.interfaces.NamingContext)
[java] | +- D (class: org.jboss.mq.SpyQueue)
[java] | +- C (class: org.jboss.mq.SpyQueue)
[java] | +- B (class: org.jboss.mq.SpyQueue)
[java] | +- A (class: org.jboss.mq.SpyQueue)
[java] | +- testQueue (class: org.jboss.mq.SpyQueue)
[java] | +- ex (class: org.jboss.mq.SpyQueue)
[java] | +- testObjectMessage (class: org.jboss.mq.SpyQueue)
[java] | +- DLQ (class: org.jboss.mq.SpyQueue)
[java] +- test (class: org.jnp.interfaces.NamingContext)
[java] | +- entity (class: org.jnp.interfaces.NamingContext)
[java] +- RMIConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
[java] +- exception (class: org.jnp.interfaces.NamingContext)
[java] +- UUIDKeyGeneratorFactory (class: org.jboss.ejb.plugins.keygenerator.uuid.UUIDKey
GeneratorFactory)
[java] +- testTCF[link -> ConnectionFactory] (class: javax.naming.LinkRef)
[java] +- ENCTests (class: org.jnp.interfaces.NamingContext)
[java] | +- ejbs (class: org.jnp.interfaces.NamingContext)
[java] +- idgen (class: org.jnp.interfaces.NamingContext)
[java] +- ejbcts2 (class: org.jnp.interfaces.NamingContext)
[java] +- commerce (class: org.jnp.interfaces.NamingContext)
[java] +- UIL2ConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
[java] +- UserTransactionSessionFactory (proxy: $Proxy10 implements interface org.jboss.tm.
usertx.interfaces.UserTransactionSessionFactory)
[java] +- UILXAConnectionFactory[link -> UIL2XAConnectionFactory]
(class: javax.naming.LinkRef)
[java] +- relation (class: org.jnp.interfaces.NamingContext)
[java] | +- manyToMany (class: org.jnp.interfaces.NamingContext)
[java] | | +- bidirectional (class: org.jnp.interfaces.NamingContext)
[java] | | +- unidirectional (class: org.jnp.interfaces.NamingContext)
[java] | +- manyToOne (class: org.jnp.interfaces.NamingContext)
[java] | | +- unidirectional (class: org.jnp.interfaces.NamingContext)
[java] | | | +- table (class: org.jnp.interfaces.NamingContext)
[java] | | | +- fk (class: org.jnp.interfaces.NamingContext)
[java] | +- oneToMany (class: org.jnp.interfaces.NamingContext)
[java] | | +- bidirectional (class: org.jnp.interfaces.NamingContext)
[java] | | | +- table (class: org.jnp.interfaces.NamingContext)
[java] | | | +- fk (class: org.jnp.interfaces.NamingContext)
[java] | | +- unidirectional (class: org.jnp.interfaces.NamingContext)
[java] | | | +- table (class: org.jnp.interfaces.NamingContext)
[java] | | | +- fk (class: org.jnp.interfaces.NamingContext)
[java] | +- oneToOne (class: org.jnp.interfaces.NamingContext)
[java] | | +- bidirectional (class: org.jnp.interfaces.NamingContext)
[java] | | | +- table (class: org.jnp.interfaces.NamingContext)
```

```
[java] | | | +- fk (class: org.jnp.interfaces.NamingContext)
[java] | | +- unidirectional (class: org.jnp.interfaces.NamingContext)
[java] | | | +- table (class: org.jnp.interfaces.NamingContext)
[java] | | | +- fk (class: org.jnp.interfaces.NamingContext)
[java] +- console (class: org.jnp.interfaces.NamingContext)
[java] { +- PluginManager (proxy: $Proxy26 implements interface
org.jboss.console.manager.PluginManagerMBean)
[java] +- HTTPXAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
[java] +- topic (class: org.jnp.interfaces.NamingContext)
[java] | +- testDurableTopic (class: org.jboss.mq.SpyTopic)
[java] | +- testTopic (class: org.jboss.mq.SpyTopic)
[java] | +- securedTopic (class: org.jboss.mq.SpyTopic)
[java] +- testQCF[link -> ConnectionFactory] (class: javax.naming.LinkRef)
[java] +- HAILXAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
[java] +- ejb (class: org.jnp.interfaces.NamingContext)
[java] | +- local (class: org.jnp.interfaces.NamingContext)
[java] | +- jca (class: org.jnp.interfaces.NamingContext)
[java] | +- remote (class: org.jnp.interfaces.NamingContext)
[java] +- cmrTransactionTest (class: org.jnp.interfaces.NamingContext)
[java] +- UserTransaction (class: org.jboss.tm.usertx.client.ClientUserTransaction)
[java] +- HTTPConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
[java] +- arrays (class: org.jnp.interfaces.NamingContext)
[java] +- RMIXAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
[java] +- HAILConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
[java] +- anotherContext (class: org.jnp.interfaces.NamingContext)
[java] | +- TopicInADifferentContext[link -> topic/myMDBTopic] (class: javax.naming.LinkRef)
[java] | +- QueueInADifferentContext[link -> queue/myMDBQueue] (class: javax.naming.LinkRef)
[java] +- psuedo-url: (class: org.jnp.interfaces.NamingContext)
[java] | +- ejb (class: org.jnp.interfaces.NamingContext)
[java] +- local (class: org.jnp.interfaces.NamingContext)
[java] +- eardeployment (class: org.jnp.interfaces.NamingContext)
[java] +- OIL2XAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
[java] +- HASessionState (class: org.jnp.interfaces.NamingContext)
[java] | +- Default (class: org.jboss.ha.hasessionstate.server.HASessionStateImpl)
[java] +- ejbcts (class: org.jnp.interfaces.NamingContext)
[java] +- UIL2XAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
[java] +- naming (class: org.jnp.interfaces.NamingContext)
[java] | +- local (class: org.jnp.interfaces.NamingContext)
[java] +- invokers (class: org.jnp.interfaces.NamingContext)
[java] | +- toki.local (class: org.jnp.interfaces.NamingContext)
[java] | | +- iiop (class: org.jboss.invocation.iiop.IIOPIInvoker)
[java] | | +- http (class: org.jboss.invocation.http.interfaces.HttpInvokerProxy)
[java] | +- 0.0.0.0 (class: org.jnp.interfaces.NamingContext)
[java] | | +- pooled (class: org.jboss.invocation.pooled.interfaces.PooledInvokerProxy)
[java] +- UILConnectionFactory[link -> UIL2ConnectionFactory] (class: javax.naming.LinkRef)
[java] +- jmx (class: org.jnp.interfaces.NamingContext)
```



```
[java] | +- invoker (class: org.jnp.interfaces.NamingContext)
[java] | | +- RMIAaptor (proxy: $Proxy25 implements interface org.jboss.
jmx.adaptor.rmi.RMIAaptor) [java] | +- rmi (class: org.jnp.interfaces.NamingContext)
[java] | | +- RMIAaptor[link -> jmx/invoker/RMIAaptor] (class: javax.naming.LinkRef)
[java] +- clustering (class: org.jnp.interfaces.NamingContext)
[java] | +- HTTPSession (proxy: $Proxy31 implements interface org.jboss.ha.
httpsession.beanimpl.interfaces.ClusteredHTTPSessionHome, interface javax.ejb.Handle)
[java] | +- LocalHTTPSession (proxy: $Proxy28 implements interface org.jboss.ha.httpsession.
beanimpl.interfaces.LocalClusteredHTTPSessionHome)
[java] +- ConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
[java] +- cmp2 (class: org.jnp.interfaces.NamingContext)
[java] | +- perf (class: org.jnp.interfaces.NamingContext)
[java] | +- audit (class: org.jnp.interfaces.NamingContext)
[java] | +- readonly (class: org.jnp.interfaces.NamingContext)
[java] | +- simple (class: org.jnp.interfaces.NamingContext)
[java] | +- lob (class: org.jnp.interfaces.NamingContext)
[java] +- v2 (class: org.jnp.interfaces.NamingContext)
[java] | +- local (class: org.jnp.interfaces.NamingContext)
[java] +- v1 (class: org.jnp.interfaces.NamingContext)
[java] | +- local (class: org.jnp.interfaces.NamingContext)
[java] </pre>
```

2.3.3 命令行方式访问 JMX

从 JBoss 3.2.1 发布版开始, JBoss 就提供了初始版本的命令行工具, 以获得和远程 JMX 服务器实例的简单交互。该工具称为“twiddle”(位于发布版的 bin 目录下), 即一个轻量级、借助于 JMX 与 MBean 交互的工具。twiddle 是命令行执行工具, 而不是一般性的命令 Shell。通过 twiddle.sh 或 twiddle.bat 脚本能够运行 twiddle。其中, 以 -h (--help) 参数形式运行 twiddle 能够获得基本语法, 以 --help-commands 参数形式运行能够获得该工具提供的具体功能。具体用法如下。

```
[nr@toki bin]$ ./twiddle.sh -h
A JMX client to 'twiddle' with a remote JBoss server.

usage: twiddle [options] <command> [command_arguments]

options:
  -h, --help                Show this help message
  --help-commands           Show a list of commands
  -H=<command>              Show command specific help
  -c=command.properties     Specify the command.properties file to use
  -D<name>[=<value>]        Set a system property
  --                        Stop processing options
  -s, --server=<url>        The JNDI URL of the remote server
  -a, --adapter=<name>      The JNDI name of the RMI adapter to use
```

```
[nr@toki bin]$ ./twiddle.sh --help-commands
twiddle.sh commands:
get          Get the values of one or more MBean attributes
invoke       Invoke an operation on an MBean
unregister   Unregister one or more MBeans
create       Create an MBean
serverinfo   Get information about the MBean server
query        Query the server for a list of matching MBeans
info         Get the metadata for an MBean
```

1. 为 twiddle 设置类路径

在 JBoss 3.2.3 中, 为使用 twiddle 工具以 RMI 方式连接到 JBoss 服务器, 开发者需要将 `jbossall-client.jar` 添加到 `JBOSS_CLASSPATH` 中。如果遇到如下错误:

```
[nr@toki bin]$ ./twiddle.sh serverinfo -list
twiddle.sh: org.jboss.util.NestedRuntimeException: - nested throwable: (javax.naming.
CommunicationException [Root exception is java.lang.ClassNotFoundException: org.jboss.proxy.Client
Container (no security manager: RMI class loader disabled)])
```

则需要开发者将上述 jar 文件添加到 `JBOSS_CLASSPATH` 中, 命令行运行如下:

```
[nr@toki bin]$ export JBOSS_CLASSPATH=../client.jbossall-client.jar
```

2. 连接 twiddle 到远程服务器

在默认情况下, twiddle 命令会连接到位于端口 1099 的 localhost, 以查询默认 RMIAdaptor 服务的 “jmx/rmi/RMIAdaptor” 绑定。其中, RMIAdaptor 服务在与 JMX 服务器通信的过程中, 充当了连接器的作用。为连接到不同的服务器/端口的组合, 开发者可以使用 `-s(--server)` 选项:

```
[nr@rubik bin]$ ./twiddle.sh -s toki serverinfo -d jboss
[nr@rubik bin]$ ./twiddle.sh -s toki:1099 serverinfo -d jboss
```

开发者使用 `-a(--adapter)` 选项能够实现到不同 RMIAdaptor 绑定的连接:

```
[nr@rubik bin]$ ./twiddle.sh -s toki -a jmx/rmi/RMIAdaptor serverinfo -d jboss
[nr@rubik bin]$ ./twiddle.sh -s toki --adapter=jmx/rmi/RMIAdaptor serverinfo -d jboss
```

3. twiddle 命令用法示例

为访问服务器的基本信息, twiddle 可使用 `serverinfo` 选项。目前支持的语法如下:

```
[nr@toki bin]$ ./twiddle.sh -H serverinfo
Get information about the MBean server

usage: serverinfo [options]

options:
  -d, --domain    Get the default domain
  -c, --count      Get the MBean count
  -l, --list       list List the MBeans
```

```
-- Stop processing options
```

```
[nr@rubik bin]$ ./twiddle.sh --server=toki serverinfo -count 385
[nr@rubik bin]$ ./twiddle.sh --server=toki serverinfo -domain jboss
```

为根据特定模式获得 MBean 名字, twiddle 可以使用 query 选项以查询服务器。目前支持如下语法:

```
[nr@rubik bin]$ ./twiddle.sh -H query
Query the server for a list of matching MBeans

usage: query [options] <query>
options:
    -c, --count          Display the matching MBean count
    --                  Stop processing options
Examples:
    query all mbeans: query '*:*'
    query all mbeans in the jboss.j2ee domain: query 'jboss.j2ee:*'
[nr@rubik bin]$ ./twiddle.sh -s toki query 'jboss:service=invoker,*'
jboss:readonly=true,service=invoker,target=Naming,type=http
jboss:service=invoker,type=jmp
jboss:service=invoker,type=httpHA
jboss:service=invoker,type=jmpha
jboss:service=invoker,type=local
jboss:service=invoker,type=pooled
jboss:service=invoker,type=iiop
jboss:service=invoker,type=http
jboss:service=invoker,target=Naming,type=http
```

为获得 MBean 的属性, twiddle 可以使用 get 选项:

```
[nr@toki bin]$ ./twiddle.sh get jboss:service=invoker,type=jmp
Get the values of one or more MBean attributes

usage: get [options] <name> [<attr>+]
    If no attribute names are given all readable attributes are gotten
options:
    --noprimary          Do not display attribute name prefixes
    --                  Stop processing options
[orb@toki bin]$ ./twiddle.sh get jboss:service=invoker,type=jmp RMIObjPort
StateString
RMIObjPort=4444
StateString=Started
[nr@toki bin]$ ./twiddle.sh get jboss:service=invoker,type=jmp
ServerAddress=0.0.0.0
StateString=Started
State=3
EnableClassCaching=false
```

```
SecurityDomain=null
RMIServerSocketFactory=null
Backlog=200
RMIObjectPort=4444
Name=JRMPInvoker
RMIClientSocketFactory=null
```

为查询 MBean 的 MBeanInfo, twiddle 可以使用 info 选项。具体如下:

```
[nr@toki bin]$ ./twiddle.sh -H info
Get the metadata for an MBean

usage: info <mbean-name>
    Use '*' to query for all attributes

[nr@toki bin]$ ./twiddle.sh info jboss:service=invoker,type=jmnp
Description: Management Bean.
+++ Attributes:
    Name: ServerAddress
    Type: java.lang.String
    Access: rw
    Name: StateString
    Type: java.lang.String
    Access: r-
    Name: State
    Type: int
    Access: r-
    Name: EnableClassCaching
    Type: boolean
    Access: rw
    Name: SecurityDomain
    Type: java.lang.String
    Access: rw
    Name: RMIServerSocketFactory
    Type: java.lang.String
    Access: rw
    Name: Backlog
    Type: int
    Access: rw
    Name: RMIObjectPort
    Type: int
    Access: rw
    Name: Name
    Type: java.lang.String
    Access: r-
    Name: RMIClientSocketFactory
    Type: java.lang.String
    Access: rw+++
```


Operations:

```
void start()
void create()
void stop()
void destroy()
```

为调用 MBean 上的操作，twiddle 可以使用 invoke 选项：

```
[nr@toki bin]$ ./twiddle.sh -H invoke
```

Invoke an operation on an MBean

```
usage: invoke [options] <query> <operation> (<arg>)*
```

options:

```
-q, --query-type[=<type>]    Treat object name as a query
--                            Stop processing options
```

query type:

```
f[first]    Only invoke on the first matching name [default]
a[all]      Invoke on all matching names
```

```
[nr@toki bin]$ ./twiddle.sh invoke jboss:service=JNDIView list true
```

```
<h1>Ejb Module: ClusteredHttpSessionEB.jar</h1>
```

```
<h2>java:comp namespace of the ClusteredHTTPSession bean:</h2>
```

```
<pre>
```

```
+ env (class: org.jnp.interfaces.NamingContext)
```

```
</pre>
```

```
<h1>java: Namespace</h1>
```

```
<pre>
```

```
+ jaas (class: javax.naming.Context)
```

```
| +- jbossmq-httpil (class: org.jboss.security.plugins.SecurityDomainContext)
```

```
| +- other (class: org.jboss.security.plugins.SecurityDomainContext)
```

```
| +- JmsXARealm (class: org.jboss.security.plugins.SecurityDomainContext)
```

```
| +- jbosstest-web (class: org.jboss.security.plugins.SecurityDomainContext)
```

```
| +- http-invoker (class: org.jboss.security.plugins.SecurityDomainContext)
```

```
| +- jbossmq (class: org.jboss.security.plugins.SecurityDomainContext)
```

```
| +- secure-jndi (class: org.jboss.security.plugins.SecurityDomainContext)
```

```
| +- HsqlDbRealm (class: org.jboss.security.plugins.SecurityDomainContext)
```

```
+ TransactionPropagationContextImporter (class: org.jboss.tm.TransactionPropagationContextImporter)
```

```
+ JmsXA (class: org.jboss.resource.adapter.jms.JmsConnectionFactoryImpl)
```

```
+ JBossCorbaNaming (class: org.omg.CosNaming.NamingContextExt)
```

```
+ DefaultDS (class: org.jboss.resource.adapter.jdbc.WrapperDataSource)
```

```
...
```

```
</pre>
```

2.3.4 使用任何协议连接到 JMX

在提供分离式 Invoker 和具备创建代理工厂能力（一定程度上）的前提下，开发者使用 InvokerAdaptorService 和代理工厂服务，并通过任意协议而暴露的 RMIAdaptor 接口或类似接口，能够实现与 JMX 服务器的通信。在“2.7 远程访问服务——分离式 Invoker”一节的内容有分离式 Invoker 和代理工厂的介绍。其中，“2.7.1 分离式 Invoker 实例：MBeanServer Invoker 适配器服务”一节有这方面的实例介绍。该实例展示了这样一种情况，即如果存在代理工厂服务，则允许客户应用基于任意协议而使用 RMIAdaptor 接口，从而实现对 MBeanServer 的访问。

2.4 将 JMX 作为微内核

当启动 JBoss 时，首当其冲要完成的就是创建 MBean 服务器实例，即 javax.management.MBeanServer。JBoss 架构中的 JMX MBean 服务器担当了微内核的作用。所有其他受管 MBean 组件都是通过注册到该 MBean 服务器，从而插入到 JBoss 中的。这里的内核仅仅表明它是框架，而不是指实际功能。实际功能由 MBean 提供，同时所有的主要 JBoss 组件都是通过 MBean 服务器相连在一起的受管 MBean。

2.4.1 启动过程

这里将讨论 JBoss 服务器的启动过程。JBoss 服务器启动过程大概是根据如下启动顺序完成启动任务的。



- (1) run 脚本使用 org.jboss.Main.main(String []) 方法入口点触发引导顺序。
- (2) Main.main 方法创建线程组“jboss”。然后，启动“jboss”线程组中的某线程。该线程调用 Main.boot 方法。
- (3) Main.boot 方法在处理 Main.main 的参数后，使用系统属性和参数指定的其他属性创建 org.jboss.system.server.ServerLoader 实例。
- (4) ServerLoader 注册 XML 解析库、jboss-jmx.jar、concurrent.jar、其他库及参数指定的类路径 (classpath)。
- (5) 将传入当前线程上下文类装载器作为 ServerLoader.load(ClassLoader) 方法的实参，从而能够创建 JBoss 服务器实例。其中，返回的服务器实例实现了 org.jboss.system.server.Server 接口。创建服务器实例具体包括如下步骤：
 - 1) 在启动过程中，使用注册于 ServerLoader 中的 jar 文件和目录的 URL，创建 java.net.URLClassLoader。该 URLClassLoader 使用传入的 ClassLoader 作为其双亲类装载器，并将其本身作为线程的上下文类装载器。

2) 这里使用 `Server` 接口的实现类名由“`jboss.server.type`”属性决定。其在默认情况下为 `org.jboss.system.server.ServerImpl`。

3) 步骤 1 创建的 `URLClassLoader` 将完成 `Server` 实现类的装载和实例化（通过无参数构建器）。其后，恢复线程的上下文类装载器，并将 `Server` 实例返回调用者。

(6) 把传递给 `ServerLoader` 构建器的属性作为 `Server.init(Properties)` 方法中 `Properties` 的实参，并初始化服务器。

(7) 使用 `Server.start()` 方法启动服务器实例。具体步骤如下：

1) 将线程的上下文类装载器设置为用于装载 `ServerImpl` 类的类装载器。

2) 使用 `MBeanServerFactory.createMBeanServer(String)` 方法创建“`jboss`”域下的 `MBeanServer`。

3) 使用 `MBeanServer` 注册 `ServerImpl` 和 `ServerConfigImpl` `MBean`。

4) 初始化统一类装载器库，并包含可选 `patch` 目录及服务器配置文件集合 `conf` 目录（如 `server/default/conf`）中的所有 `jar` 文件。对于每个 `jar` 和目录，都会创建新的 `org.jboss.mx.loading.UnifiedClassLoader`，并在统一库中注册。其中，JBoss 会从创建的 `UnifiedClassLoader` 中选择一个作为线程的上下文类装载器。因此，线程的上下文类装载器能够使用到所有的 `UnifiedClassLoader`。

5) 创建 `org.jboss.system.ServiceController` `MBean`。`ServiceController` 管理 JBoss `MBean` 服务的生命周期。“2.4.2 JBoss `MBean` 服务”一节将深入讨论 JBoss `MBean` 服务。

6) 创建并启动 `org.jboss.deployment.MainDeployer`。`MainDeployer` 管理部署单元的依赖性，并保证部署单元能够分发到正确的部署器。

7) 创建并启动 `org.jboss.deployment.JARDeployer`。`JARDeployer` 处理简单 `jar` 库的部署。

8) 创建并启动 `org.jboss.deployment.SARDeployer`。`SARDeployer` 处理 JBoss `MBean` 服务的部署。

9) 调用 `MainDeployer`，以完成当前服务器文件集合中 `conf/jboss-service.xml` 定义的服务。

10) 恢复线程的上下文类装载器。

JBoss 服务器启动初期，仅仅是作为 `JMX` `MBean` 服务器的容器。然后，装载 `jboss-service.xml` `MBean` 配置文件中定义的各种个性化服务。这里的 `jboss-service.xml` 文件位于从 `run` 命令行传入的特定配置集合（`conf` 目录）中。由于 `MBean` 定义了 JBoss 服务器实例的功能，因此理解核心 JBoss `MBean` 是如何开发的及如何使用 `MBean` 集成现有服务到 JBoss 中显得很重要。这也是下节将要讨论的主要内容。

2.4.2 JBoss `MBean` 服务

正如大家所看到的一样，JBoss 依赖 `JMX` 装载 `MBean` 服务。而这些 `MBean` 服务组成了具体服务器实例的差异性。标准 JBoss 发布版提供的、所有绑定的功能都是基于 `MBean` 的。为 JBoss 服务器添加新服务的最好办法是，开发自己的 `JMX` `MBean`。

目前存在两种 MBean。一种独立于 JBoss 服务,另一种依赖于 JBoss 服务。独立于 JBoss 的 MBean 还处于试用阶段。它可以通过 JMX 规范开发,并在 `deploy/user-service.xml` 文件中添加 `mbean` 标签,从而添加到 JBoss 服务器中。对于依赖于 JBoss 服务器的 MBean (比如命名),则需要开发者遵循 JBoss 服务模式才能开发该类型的 MBean。JBoss MBean 服务模式由一套用于提供状态变更通知的生命周期操作组成。当 MBean 服务创建、启动、停止及销毁自身时,这些通知便需要告之其他 MBean 服务。MBean 服务生命周期的管理由如下 3 个 JBoss MBean 负责: `SARDeployer`、`ServiceConfigurator` 及 `ServiceController`。

1. SARDeployer MBean

JBoss 借助于自定义 MBean 实现 MBean 服务部署的管理。其中,自定义 MBean 能够从标准的 JMX MLet 配置文件读取 XML 变量。自定义 MBean 由 `org.jboss.deployment.SARDeployer` 类实现。`SARDeployer` MBean 是 JBoss 启动时装入的,它也是 JBoss 引导过程的组成部分。缩写 SAR 代表服务存档 (Service Archive)。

`SARDeployer` 处理服务存档。服务存档可能是后缀为 `.sar` 的 jar 文件,并含有 `META-INF/jboss-service.xml` 描述符。它也可能是匹配 “*-service.xml” 命名模式的单独 XML 配置符。图 2-16 给出了服务描述符的 DTD 结构。

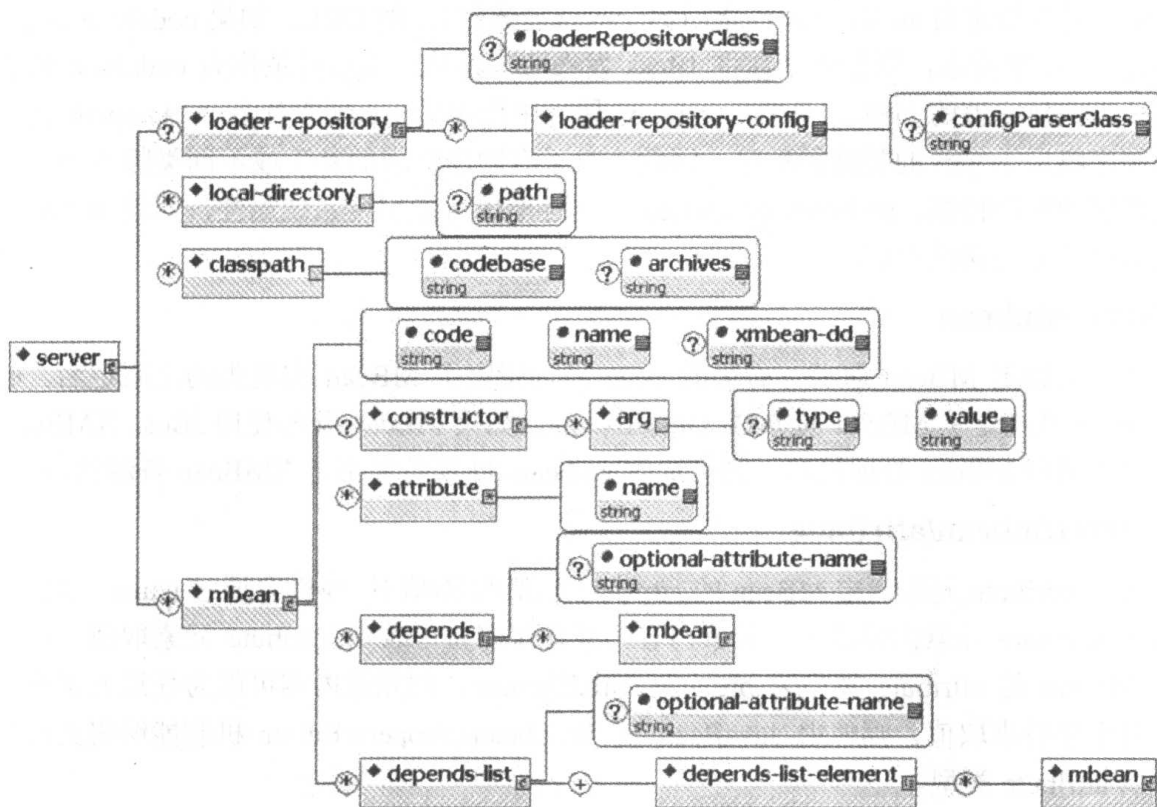


图 2-16 供 SARDeployer 分析、用于 MBean 服务描述符的 DTD

该 DTD 包含的元素如下。

● server/loader-repository

该元素用于指定 `UnifiedLoaderRepository` MBean 的名字。其中,该 MBean 为 SAR 提供部署在 SAR 中 SAR 级别的范围类。该元素值指定了惟一的 JMX ObjectName 字符串。

同时，该元素还可能包含 `loader-repository-config` 元素，以提供其他的配置信息。可选 `loaderRepositoryClass` 属性用于指定装载库实现类的全限定名。其在默认情况下的取值为 `org.jboss.mx.loading.HeirarchicalLoaderRepository3`。

● `server/loader-repository/loader-repository-config`

该可选元素为 `loaderRepositoryClass` 属性提供其他所需配置。可选 `configParserClass` 属性指定 `org.jboss.mx.loading.LoaderRepositoryFactory.LoaderRepositoryConfigParser` 接口实现指定全限定名，用于分析 `loader-repository-config` 元素内容。

● `server/local-directory`

该元素指定部署存档中的路径信息，将包含在路径中的内容复制到 `server/<config>/db` 目录中供 MBean 使用。该 `path` 属性是部署存档中的入口名。

● `server/classpath`

该元素给出应该部署在 MBean 中的一个或多个外部 jar 文件。其中，可选属性 `archives` 指定以逗号隔开的、待装载的 jar 名列表，或者使用通配符 “*” 以装载所有 jar。该通配符仅对文件 URL 有效，但如果 Web 服务器支持 WebDAV 协议，则也支持 HTTP URL。`codebase` 属性指定由 `archives` 属性指明的、应该装载的 jar 的 URL。如果 `codebase` 是 path，而不是 URL 字符串，则服务器会将 JBoss 发布服务器/`<config>` 目录作为 `codebase` 值的相对路径以构建完整的 URL。其中，`archives` 属性中指定的 jar 顺序及多个 `classpath` 元素形成的顺序将用于 jar 的类路径顺序。因此，对于有顺序要求的补丁或不同类版本而言，必须注意这种顺序特征。`archives` 和 `codebase` 属性取值可能引用系统属性，即使用 “\${x}” 模式以代替系统属性 “x”。

● `server/mbean`

该元素给定 MBean 服务。必需的 `code` 属性指定了 MBean 实现类的全限定名。必需的 `name` 属性指定了 MBean 的 JMX ObjectName。如果 MBean 服务使用 JBoss XMBean 描述符定义模型 MBean 管理接口，则可选的 `xmbean-dd` 属性将指定 XMBean 资源路径。

● `server/mbean/attribute`

各个 `attribute` 元素指定 MBean 的 `attribute` 元素的名/值对。属性名通过 `name` 属性标识，值通过 `attribute` 元素内容给出。元素内容可能通过字符串表示 `attribute` 元素取值。或者，如果 MBean 的 `attribute` 类型为 `org.w3c.dom.Element`，则元素内容可能为任意元素和子元素。对于字符串取值，则使用 JavaBean 的 `java.beans.PropertyEditor` 机制能够将其转换为相应的 `attribute` 类型。

自从 JBoss 3.0.5 发布版开始，属性的字符串取值可以通过 “\${x}” 模式引用系统属性。存在这种情形时，属性值则是 `System.getProperty(“x”)` 返回的结果，如果该属性不存在则返回 `null`。

● `server/mbean/depends` 和 `server/mbean/depends-list`

这些元素指定了该 MBean 与其依赖的其他 MBean 的依赖性关系。“4. 指定服务依赖性” 一节有该方面的详细阐述。请注意，依赖的 MBean 元素有可能还嵌入其他 MBean。

当 SARDeployer 需要部署 MBean 服务时, 它需要完成如下几个步骤。图 2-17 展示的序列图描述了这几个步骤。

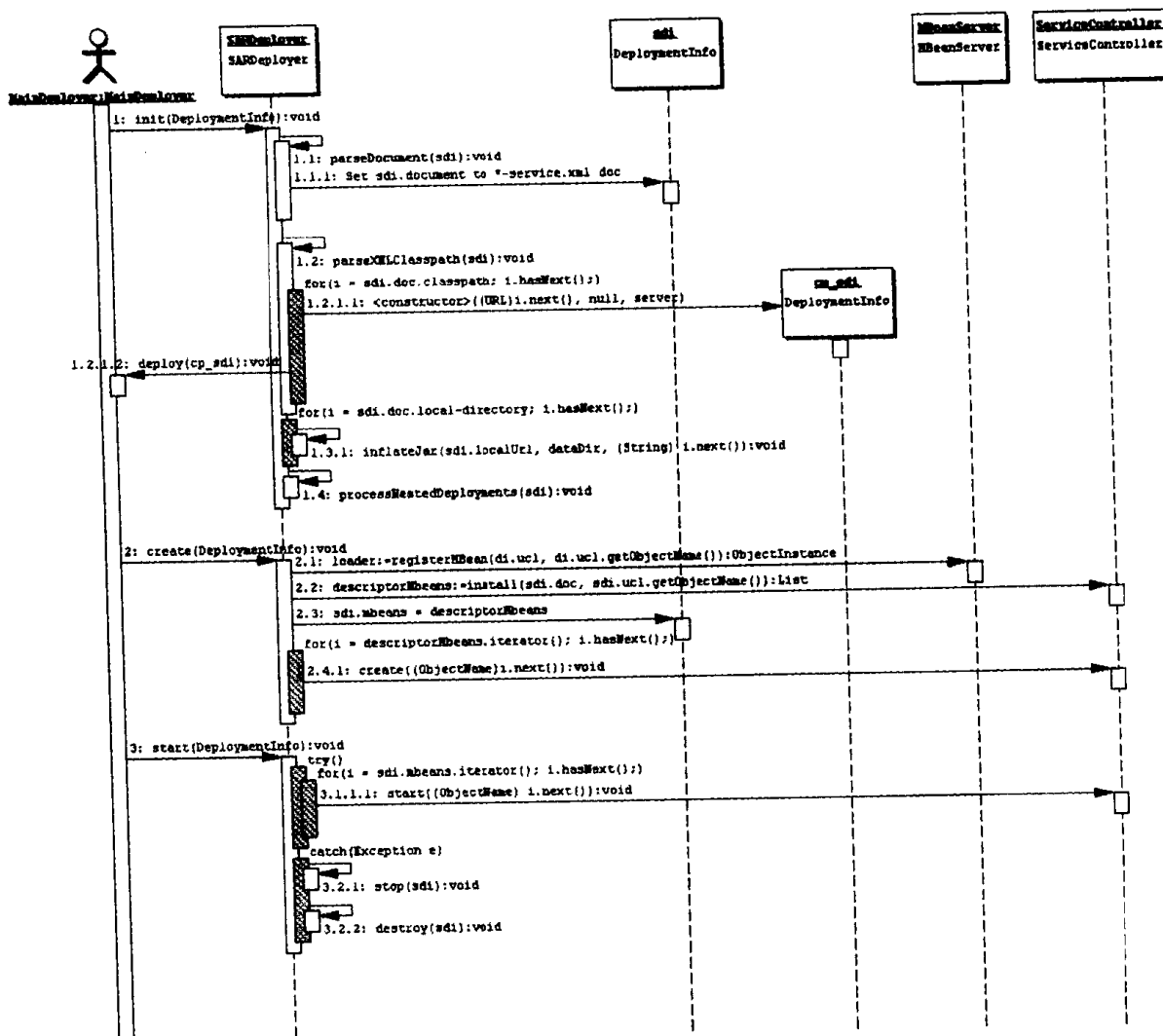


图 2-17 序列图 (SARDeployer 在启动 JBoss MBean 服务过程中完成的主要行为)

图 2-17 给出的内容如下。



- (1) 前缀为 1.1 的方法用于装载和分析 XML 服务描述符。
- (2) 前缀为 1.2 的方法用于处理服务描述符中的各个 classpath 元素, 以创建单独的部署。其中, 这些部署使得注册到统一装载库的 UnifiedClassLoader 能够使用到各个 classpath 元素中的 jar 和目录。
- (3) 前缀为 1.3 的方法用于处理服务描述符中的各个 local-directory 目录, 即将 path 属性所描述的内容拷贝到 server/<config>/db 目录。
- (4) 方法 1.4 用于处理嵌入在服务中的各个部署单元。SARDeployer MBean 将创建子部署应用, 并将其添加到 (服务部署信息) 子部署列表中。

(5) 在方法 2.1 中, SAR 部署单元的 UnifiedClassLoader 注册到 MBeanServer 上, 使得其能够用于装载 SAR MBean。

(6) 方法 2.2 针对配置符中的各个 mbean 元素, 创建相应的实例, 其属性初始化为服务描述符中给定的值。这些操作都是通过调用方法 ServiceController.install 完成的。

(7) 方法 2.4.1 针对创建的各个 MBean 实例, 分别获得其 JMX ObjectName, 并将创建服务生命周期的任务请求发送给 ServiceController。ServiceController 处理 MBean 服务的依赖性。仅当服务依赖性得到满足时, 才会调用服务的 create 方法。

(8) 前缀为 3.1 的方法用于启动服务描述符中的各个 MBean 服务。对于各个创建的 MBean 实例, 该方法能够获得各自的 JMX ObjectName, 并将启动服务生命周期的任务请求发送给 ServiceController。ServiceController 处理 MBean 服务的依赖性。仅当服务依赖性得到满足时, 才会调用服务的 start 方法。

2. Service 生命周期接口

JMX 规范并没有为 MBean 定义任何类型的生命周期或依赖性管理。但是, JBoss 引入的 ServiceController MBean 却具备这些功能。JBoss MBean 对 JMX MBean 的扩展主要在如下方面, 即降低构建 MBean 实例和其服务生命周期之间的耦合度。为实现任何类型的依赖性管理, 必须借助于这一特性。比如, 如果开发的 MBean 需要操作 JNDI 命名服务, 则当服务依赖性得到满足时, 必须将该信息告知该 MBean。如果 MBean 构建器是惟一的生命周期事件, 则完成上述操作太困难了, 甚至不可能实现。因此, JBoss 引入了服务生命周期接口来描述事件, 使得它能够管理 MBean 的行为。列表 2-14 给出了 org.jboss.system.Service 接口。

列表 2-14 org.jboss.system.Service 接口

```
package org.jboss.system;

public interface Service
{
    public void create() throws Exception;
    public void start() throws Exception;
    public void stop();
    public void destroy();
}
```

ServiceController MBean 在服务生命周期的合适时机将调用 Service 接口的方法。有关这些方法的进一步细节, 请参考 ServiceController 节内容。



J2EE 管理规范请求 (JSR 77, <http://jcp.org/jsr/detail/77.jsp>) 引入的状态管理中包含了启动和停止生命周期方面的内容。一旦该规范请求的最终版发布后, JBoss 有可能支持 JSR 77 扩展, 即基于服务生命周期的实现。从 JBoss 3.2.0 发布版开始, JBoss 已经支持 JSR 77 管理对象及其大部分规范内容, 但还未提供生命周期操作支持。

3. ServiceController MBean

JBoss 借助于自定义 `org.jboss.system.ServiceController` MBean 实现 MBean 之间的依赖性管理。SARDeployer 将 MBean 服务的初始化、创建、启动、停止及销毁工作委派给 ServiceController。图 2-18 展示了一个序列图，重点描述 SARDeployer 和 ServiceController 之间的交互。

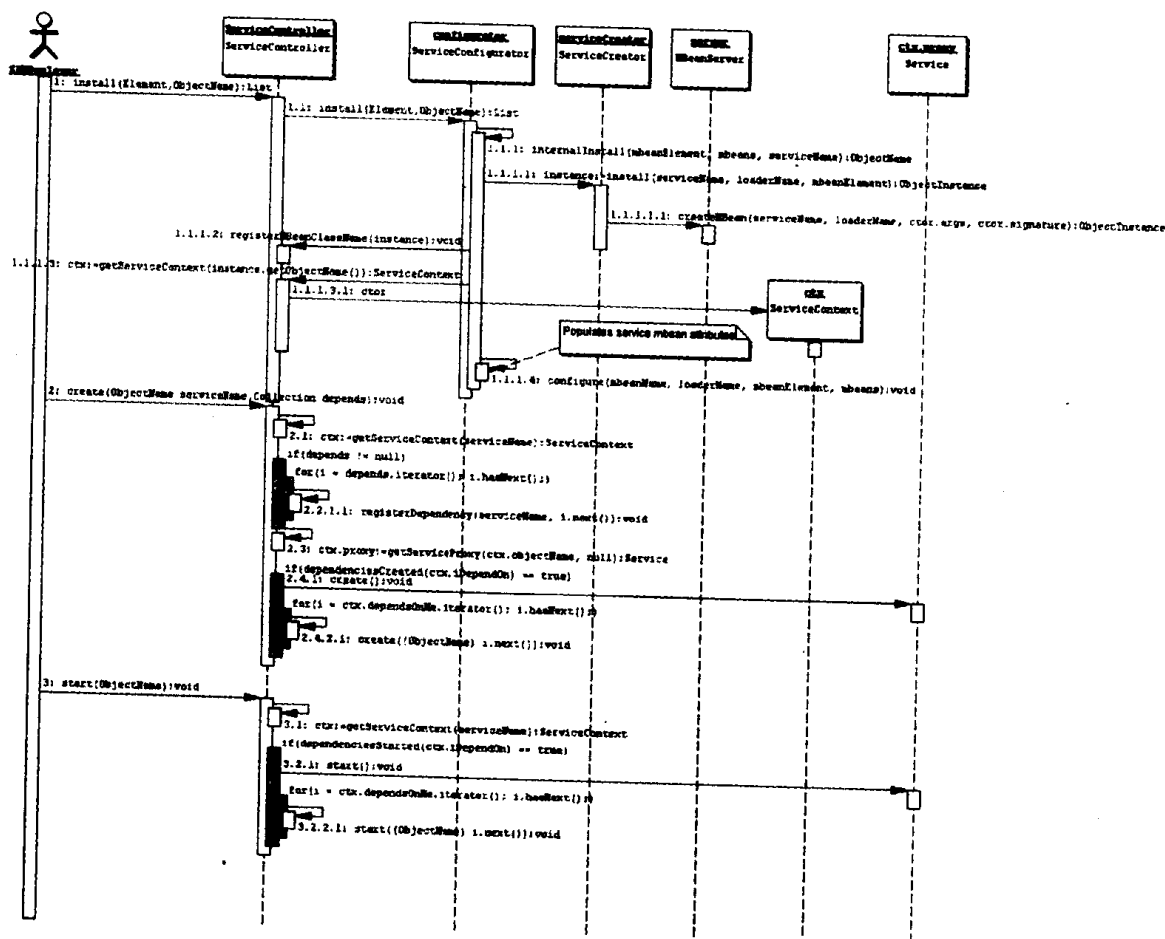


图 2-18 序列图 (为启动某服务触发 SARDeployer 和 ServiceController 之间的交互)

ServiceController MBean 提供了 4 个重要方法, 用于管理服务生命周期, 即 `create`、`start`、`stop` 及 `destroy` 方法。

- `create(ObjectName)` 方法

无论何时, 只要出现的某事件会影响指定服务的状态时, 就可能触发 `create(ObjectName)` 方法的调用。通过 SARDeployer 的显式调用, 比如通知某个新类, 或告知某 MBean 服务到达了其创建状态, 就会触发 `create(ObjectName)` 方法的调用。

当调用某服务的 `create` 方法时, 该服务所依赖的所有其他服务的 `create` 方法也将被触发。这使得 MBean 有机会检查所要求的 MBean 或资源是否存在。但此时还是不能够使用其他的 MBean 服务, 因为对于大部分 JBoss MBean 服务而言, 在没有借助于 `start` 方法启动它们之前, 并不是 MBean 服务的所有功能都可用。据此, 服务实现常常是不实现 `create` 方法, 而仅仅实现 `start` 方法, 因为 `start` 方法使得 MBean 服务能提供所有功能的入口。

- start(ObjectName)方法

无论何时，只要出现的某事件会影响指定服务的状态时，就可能触发 start(ObjectName)方法的调用。通过 SARDeployer 的显式调用，比如通知某个新类，或告知某 MBean 服务到达了其启动状态，就会触发 start(ObjectName)方法。

当调用某服务的 start 方法时，该服务所依赖的所有其他服务的 start 方法也将被触发。既然该 MBean 服务依赖的所有其他 MBean 服务都已经创建和启动完成，则 start 方法调用标识服务的所有功能处于可操作状态。

- stop(ObjectName)方法

无论何时，只要出现的某事件会影响指定服务的状态时，就可能触发 stop(ObjectName)方法的调用。通过 SARDeployer 的显式调用，比如通知某个新类，或告知某 MBean 服务到达了其停止状态，就会触发 stop(ObjectName)方法。

- destroy(ObjectName)方法

无论何时，只要出现的某事件会影响指定服务的状态时，就可能触发 destroy(ObjectName)方法的调用。通过 SARDeployer 的显式调用，比如通知某个新类，或告知某 MBean 服务到达了其销毁状态，就会触发 destroy(ObjectName)方法。

在通常情况下，Service 接口实现并不会实现 destroy 方法，而只是简单地实现 stop 方法。甚至，如果 MBean 服务没有状态或资源需要清除，stop 和 destroy 方法都不会实现。

4. 指定服务依赖性

为指定 MBean 服务依赖的其他 MBean 服务，开发者必须在服务描述符的 mbean 元素中声明这种依赖性，即通过 depends 和 depends-list 元素能够达到此目的。这两个元素的区别之一在于 optional-attribute-name 属性的用法。如果通过单值属性跟踪依赖的 ObjectName，则应该使用 depends 元素。如果使用 java.util.List 兼容的属性跟踪依赖的 ObjectName，则应该使用 depends-list 元素。如果只是声明依赖性，而不注重绑定到 MBean 属性的相关服务 ObjectName，则建议使用最容易使用的元素。列表 2-15 给出了这方面实例描述片段，即阐述依赖性相关元素的使用。

列表 2-15 阐述 depends 和 depends-list 元素用法的服务描述符片段

```
<mbean code="org.jboss.mq.server.jmx.Topic"
  name="jms.topic:service=Topic,name=testTopic">
  <!-- Declare a dependency on the "jboss.mq:service=DestinationManager" and
    bind this name to the DestinationManager attribute -->
  <depends optional-attribute-name="DestinationManager">
    jboss.mq:service=DestinationManager
  </depends>
  <!-- Declare a dependency on the "jboss.mq:service=SecurityManager" and
    bind this name to the SecurityManager attribute -->
  <depends optional-attribute-name="SecurityManager">
    jboss.mq:service=SecurityManager
  </depends>
  ...
  <!-- Declare a dependency on the "jboss.mq:service=CacheManager" without
```

```

any binding of the name to an attribute-->
<depends>jboss.mq:service=CacheManager</depends>
</mbean>
<mbean code="org.jboss.mq.server.jmx.TopicMgr"
name="jboss.mq.destination:service=TopicMgr">
<!-- Declare a dependency on the given topic destination mbeans and
bind these names to the Topics attribute -->
<depends-list optional-attribute-name="Topics">
<depends-list-element>jms.topic:service=Topic,name=A</depends-list-element>
<depends-list-element>jms.topic:service=Topic,name=B</depends-list-element>
<depends-list-element>jms.topic:service=Topic,name=C</depends-list-element>
</depends>
</mbean>

```

`depends` 和 `depends-list` 元素的另一个区别在于, `depends` 元素的取值可能是完整的 MBean 服务配置, 而不仅仅是服务的 ObjectName。列表 2-16 摘了 `hsqldb-service.xml` 中的一段作为实例。该列表通过使用嵌入的 MBean 作为 `depends` 元素取值以定义 `org.jboss.resource.connectionmanager.RARDeployment` 服务配置。它表明, `org.jboss.resource.connectionmanager.LocalTxConnectionManager` MBean 服务依赖于 `RARDeployment` 服务。其中, 如 ObjectName, 即 “`jboss.jca:service=LocalTxDS,name= hsqldbDS`” 将被绑定到 `LocalTxConnectionManager` 类的 `ManagedConnectionFactoryName` 属性上。

列表 2-16 使用 `depends` 元素以指定含有依赖性的完整服务配置实例

```

<mbean code="org.jboss.resource.connectionmanager.LocalTxConnectionManager"
name="jboss.jca:service=LocalTxCM,name=hsqldbDS">
<depends optional-attribute-name="ManagedConnectionFactoryName"> <!--embedded mbean-->
<mbean code="org.jboss.resource.connectionmanager.RARDeployment"
name="jboss.jca:service=LocalTxDS,name=hsqldbDS">
<attribute name="JndiName">DefaultDS</attribute>
<attribute name="ManagedConnectionFactoryProperties">
<properties>
<config-property name="ConnectionURL" type="java.lang.String">
jdbc:hsqldb:hsqldb://localhost:1476
</config-property>
<config-property name="DriverClass" type="java.lang.String"> org.hsqldb.jdbcDriver
</config-property>
<config-property name="UserName" type="java.lang.String">sa </config-property>
<config-property name="Password" type="java.lang.String"/> </properties>
</attribute>
...
</mbean>
...
</mbean>

```

5. 识别不合要求的依赖性

ServiceController MBean 提供了两个操作，以帮助调试由于不合要求的依赖性而未运行 MBean 的情形。其一为 listIncompleteDeployed 操作。它为不处于运行态（RUNNING）的 MBean 服务返回 org.jboss.system.ServiceContext 对象的 java.util.List 集合。

其二为 listWaitingMBeans 操作。该操作返回未成功部署的 MBean 服务的 JMX ObjectName 的 java.util.List 集合。这些未部署的 MBean 服务的触发原因是寻找不到 code 属性指定的类。

6. 组件的热部署（URLDeploymentScanner）

URLDeploymentScanner MBean 服务为 JBoss 提供了热部署能力。该服务监视一个或多个 URL 中的可部署存档，一旦有新的存档或存档发生变动，该服务便会部署它们。同时，如果删除某已部署的存档，该 MBean 服务也将卸载它。该 MBean 服务提供的可配置属性如下：

● URLs

通过“,”隔开的 URL 字符串列表，该 MBean 服务用于定位其监视的位置。如果字符串不能够对应于有效 URL，则 MBean 服务视其为文件路径。其中，文件路径是相对于服务器的 home URL 解析的。比如，默认配置文件集合可以用 JBOSS_DIST/server/default 表示。如果 URL 表示某文件，则该 MBean 服务部署该文件，并且负责监视其随后的更新和删除操作。如果 URL 以“/”结束表示某目录，则 MBean 服务将该目录的内容作为可部署集合对待，并且该 MBean 服务负责监视其随后的更新或删除操作。以“/”为结束标志的 URL 来识别目录的方法遵循了 RFC2518 协议。同时，这种办法还使得区分集合和那些包含简单、展开存档的目录成为可能。

URLs 属性的默认值为“deploy/”。其含义为，任何存放在 server/<name>/deploy 目录下的 SAR、EAR、JAR、WAR、RAR 等存档都将自动部署并且被 JBoss MBean 服务监视。比如，URLs 实例如下：

- “deploy/”扫描\${jboss.server.url}/deploy/，其位置是本地还是远程需要取决于用于引导服务器的 URL。
- “\${jboss.server.home.dir}/deploy/”扫描{jboss.server.home.dir}/deploy，其位置总是在本地。
- “file:/var/opt/myapp.ear”，即从本地部署 myapp.ear。
- “file:/var/opt/apps/”，即扫描指定的目录。
- “http://www.test.com/netboot/myapp.ear”，即从远程部署 myapp.ear。
- “http://www.test.com/netboot/apps/”，即使用 WebDAV 扫描指定的远程位置。当然，仅当远程 HTTP 服务器支持 WebDAV PROPFIND 命令时，这种方式才是有效的。

● ScanPeriod

指定扫描器线程运行的时间间隔（以毫秒为单位）。默认取值为 5 000 毫秒（5 秒）。

● URLComparator

URLComparator 属性用于指定实现了 java.util.Comparator 的类名，即指定某目标扫描

目录中部署单元的部署顺序。该实现必须能够比较传递给 `compare` 方法的两个 `java.net.URL` 对象。其默认设置为 `org.jboss.deployment.DeploymentSorter` 类，即通过目标部署 URL 的后缀名顺序部署它们。它指定的后缀顺序为“sar”、“service.xml”、“rar”、“jar”、“war”、“wsr”、“ear”和“zip”。

另一 `Comparator` 实现是 `org.jboss.deployment.scanner.PrefixDeploymentSorter` 类。它是基于数字前缀对 URL 进行排序的。JBoss 将前缀中的数字转换为 `int` 型（忽略前面的 0），即小数字排在大数字的前面。其中，如果前缀不是以数字开头的部署单元，则它们将在以数字开头的部署单元之后完成部署。对于前缀值相同的部署单元，则通过 `DeploymentSorter` 中提供的逻辑进一步排序。

● Filter

实现了 `java.io.FileFilter` 的类名，用于过滤目标扫描目录的内容。任何不被该过滤器接受的文件将不会部署在 JBoss 服务器上。默认的过滤器实现类为 `org.jboss.deployment.scanner.DeploymentFilter`。该默认实现拒绝如下模式：

"#*", "%*", ".*", ".*_", "\$*", ".*#", ".*\$", ".*%", ".*.BAK", ".*.old", ".*.orig", ".*.rej", ".*.bak", ".*.v", ".*~", ".make.state", ".nse_depinfo", "CVS", "CVS.admin", "RCS", "RCSLOG", "SCCS", "TAGS", "core", "tags"。

● RecursiveSearch

该属性表明是否将子目录作为可部署内容。如果属性值为 `false`，则对于目录名不包含“.”的部署子目录而言，JBoss 视其为具有嵌入子部署单元的展开 jar 包。如果属性值为 `true`，则部署子目录只不过是可部署内容的分支（`grouping`）。这两种视图的不同之处表明，它们与 JBoss 支持的深度优先部署模型有关。`false` 设置，即将目录视为具有嵌入内容的展开 jar 包，一旦部署完成该 jar 目录，将触发其嵌入内容的部署。`true` 设置简单地忽略了目录，并将其内容添加到可部署单元列表中，最后基于本文上述阐述的过滤器逻辑计算顺序。`RecursiveSearch` 默认取值为 `true`。然而，有一点请开发者注意，即 JBoss 3.2.1 发布版中 `default` 配置的 `RecursiveSearch` 取值为 `false`。

● Deployer

`Deployer` 属性指实现 `org.jboss.deployment.Deployer` 接口操作的 MBean JMX `ObjectName` 字符串。其默认设置是在引导启动过程中创建的 `MainDeployer`。

2.4.3 开发 JBoss MBean 服务

如果自定义 MBean 服务依赖其他 MBean 服务，并将其集成到 JBoss 服务器中需要使用接口模式，即 `org.jboss.system.Service`。由于 JMX 规范并没有提供依赖性管理，因此当自定义 MBean 服务依赖于其他 MBean 服务时，开发者通过 `javax.management.MBeanRegistration` 接口方法是无法完成任何依赖服务的初始化工作的。反而，开发者必须使用 `Service` 接口的 `create` 和（或）`start` 方法管理依赖性状态信息。通过如下任一种办法都能够达到此目的：

- 对调用 MBean 的 MBean 接口添加 `Service` 中的任何方法。这种方式使得 MBean 实现避免了对 JBoss 具体接口的依赖。

- 为 MBean 接口扩展 org.jboss.system.Service 接口。
- 为 MBean 接口扩展 org.jboss.system.ServiceMBean 接口。ServiceMBean 是 org.jboss.system.Service 的子接口，它在其基础上添加了 String getName()、int getState() 及 String getStateString() 方法。

选择哪种方法取决于开发者是否打算将待开发的 MBean 依赖于 JBoss 具体代码。如果不打算依赖于 JBoss 代码，则请开发者选择上述第一种方式。如果开发者不关注对 JBoss 类的依赖性，则最简单的办法是：将 MBean 接口继承于 org.jboss.system.ServiceMBean 接口，将 MBean 实现类继承于抽象 org.jboss.system.ServiceMBeanSupport 类。ServiceMBeanSupport 类实现了 org.jboss.system.ServiceMBean 接口。ServiceMBeanSupport 在提供 create、start、stop 及 destroy 方法实现的同时，还集成了日志和 JBoss 服务状态管理跟踪功能。这些方法将各自待完成的具体工作分别委派给子类的 createService、startService、stopService 及 destroyService 方法。在实现 ServiceMBeanSupport 接口时，开发者需要根据实际需求重载 createService、startService、stopService 及 destroyService 中的一个或多个方法。

1. 标准 MBean 实例

这里开发一个简单的 MBean，它将某 HashMap 绑定到 JBoss JNDI 命名空间中。这里的 JBoss JNDI 命名空间的位置信息由待开发 MBean 的 JndiName 属性决定。本文开发这样一个 MBean，能够为开发者阐述创建自定义 MBean 需要完成哪些工作。由于该 MBean 使用了 JNDI，因此它依赖于 JBoss 命名服务 MBean。另外，当命名服务可用时，必须通过 JBoss MBean 服务模式通知该 MBean。

本文待开发的 MBean 称为 JNDIMap。列表 2-17 给出了基于 Service 接口方法模式的 JNDIMapMBean 接口及其 JNDIMap 实现类的 Version 1 版本。该版接口利用上述的第一种办法，即将 Service 接口中完成启动任务的方法正确地合并到 JNDIMapMBean，而不使用 JBoss 具体相关的接口。该接口包含的 Service.start 方法能够完成所需服务的启动任务，Service.stop 方法能够完成服务的清除工作。

列表 2-17 基于 Service 接口方法模式的 JNDIMapMBean 接口及其实现

```
package org.jboss.chap2.ex1;

// The JNDIMap MBean interface
import javax.naming.NamingException;

public interface JNDIMapMBean
{
    public String getJndiName();
    public void setJndiName(String jndiName) throws NamingException;
    public void start() throws Exception;
    public void stop() throws Exception;
}

package org.jboss.chap2.ex1;
```



```
// The JNDIMap MBean implementation
import java.util.HashMap;
import javax.naming.InitialContext;
import javax.naming.Name;
import javax.naming.NamingException;
import org.jboss.naming.NonSerializableFactory;

public class JNDIMap implements JNDIMapMBean
{
    private String jndiName;
    private HashMap contextMap = new HashMap();
    private boolean started;
    public String getJndiName()
    {
        return jndiName;
    }
    public void setJndiName(String jndiName) throws NamingException
    {
        String oldName = this.jndiName;
        this.jndiName = jndiName;
        if( started )
        {
            unbind(oldName);
            try
            {
                rebind();
            }
            catch(Exception e)
            {
                NamingException ne = new NamingException("Failed to update jndiName");
                ne.setRootCause(e);
                throw ne;
            }
        }
    }
    public void start() throws Exception
    {
        started = true;
        rebind();
    }
    public void stop()
    {
        started = false;
        unbind(jndiName);
    }
}
```

```
private void rebind() throws NamingException
{
    InitialContext rootCtx = new InitialContext();
    Name fullName = rootCtx.getNameParser("").parse(jndiName);
    System.out.println("fullName="+fullName);
    NonSerializableFactory.rebind(fullName, contextMap, true);
}
private void unbind(String jndiName)
{
    try
    {
        InitialContext rootCtx = new InitialContext();
        rootCtx.unbind(jndiName);
        NonSerializableFactory.unbind(jndiName);
    }
    catch(NamingException e)
    {
        e.printStackTrace();
    }
}
```

基于 `ServiceMBean` 接口和 `ServiceMBeanSupport` 类的 `JNDIMapMBean` 接口及其 `JNDIMap` 实现类的 Version 2 版本，见列表 2-18 所示。该版的实现类继承于 `ServiceMBeanSupport` 类，并重载了 `startService` 和 `stopService` 方法。与此同时，`JNDIMapMBean` 也实现了抽象方法 `getName`，以返回 `MBean` 的描述性名称。另外，`JNDIMapMBean` 继承于 `org.jboss.system.ServiceMBean` 接口。由于其 `Service` 生命周期方法继承于 `ServiceMBean` 接口，因此 `JNDIMapMBean` 仅仅提供了 `JndiName` 属性的 `setter` 和 `getter` 方法。这正是“2.4.3 开发 JBoss MBean 服务”节阐述的第三种 `MBean` 实现方法。列表 2-18 中粗体部分标出了与列表 2-17 不同的地方。

列表 2-18 基于 `ServiceMBean` 接口和 `ServiceMBeanSupport` 类的 `JNDIMap MBean` 接口和实现

```
package org.jboss.chap2.ex2;

// The JNDIMap MBean interface
import javax.naming.NamingException;

public interface JNDIMapMBean extends org.jboss.system.ServiceMBean
{
    public String getJndiName();
    public void setJndiName(String jndiName) throws NamingException;
}

package org.jboss.chap2.ex2;
// The JNDIMap MBean implementation
```

```
import java.util.HashMap;
import javax.naming.InitialContext;
import javax.naming.Name;
import javax.naming.NamingException;
import org.jboss.naming.NonSerializableFactory;

public class JNDIMap extends org.jboss.system.ServiceMBeanSupport
    implements JNDIMapMBean
{
    private String jndiName;
    private HashMap contextMap = new HashMap();

    public String getJndiName()
    {
        return jndiName;
    }

    public void setJndiName(String jndiName) throws NamingException
    {
        String oldName = this.jndiName;
        this.jndiName = jndiName;
        if( super.getState() == STARTED )
        {
            unbind(oldName);
            try
            {
                rebind();
            }
            catch(Exception e)
            {
                NamingException ne = new
                    NamingException("Failed to update jndiName");
                ne.setRootCause(e);
                throw ne;
            }
        }
    }

    public void startService() throws Exception
    {
        rebind();
    }

    public void stopService()
    {
        unbind(jndiName);
    }
}
```

```

private void rebind() throws NamingException
{
    InitialContext rootCtx = new InitialContext();
    Name fullName = rootCtx.getNameParser("").parse(jndiName);
    log.info("fullName="+fullName);
    NonSerializableFactory.rebind(fullName, contextMap, true);
}
private void unbind(String jndiName)
{
    try
    {
        InitialContext rootCtx = new InitialContext();
        rootCtx.unbind(jndiName);
        NonSerializableFactory.unbind(jndiName);
    }
    catch(NamingException e)
    {
        log.error("Failed to unbind map", e);
    }
}
}

```

examples/src/main/org/jboss/chap2/{ex1,ex2} 目录存放了上述 MBean 源代码及相应的服务描述符。

列表 2-19 给出了实例 1 的服务描述符和客户端代码实例片段。JNDIMap MBean 将 HashMap 对象绑定在 JNDI 名字 “inmemory/maps/MapTest” 下。同时，客户端代码实例演示了如何从 “inmemory/maps/MapTest” 位置获得 HashMap 对象。

列表 2-19 实例 1 的服务描述符和客户端代码实例片段

```

<!-- The SAR META-INF/jboss-service.xml descriptor -->
<server>
    <mbean code="org.jboss.chap2.ex1.JNDIMap" name="chap2.ex1:service=JNDIMap">
        <attribute name="JndiName">inmemory/maps/MapTest</attribute>
        <depends>jboss:service=Naming</depends>
    </mbean>
</server>

// Sample lookup code
InitialContext ctx = new InitialContext();
HashMap map = (HashMap) ctx.lookup("inmemory/maps/MapTest");

```

2. XMBean 实例

这里将开发上面给出的 JNDIMap MBean 的其他实现方式。其中，该实例将使用 JBoss XMBean 框架暴露其管理元数据。除没有实现任何具体的管理相关接口外，待开发的核心受管组件和 JNDIMap 类的核心代码完全一样。在这里将阐述如下几方面标准 MBean 所不

具备的特性:

- 对属性和操作添加丰富的描述信息。
- 暴露通知信息。
- 添加属性的持久化。
- 为通过类型化接口实现远程访问和安全性添加自定义拦截器。

(1) 版本 1, 有注解的 JNDIMap XMBean

接下来, 本文以 JNDIMap 的标准 MBean 版本为出发点, 并向其添加属性、操作及参数的描述信息, 最终开发者能够获得一个简单的 JNDIMap XMBean 版本实现。列表 2-20 展示了 jboss-service.xml 和 jndimap-xmbean1.xml XMBean 描述符。通过本书光盘目录, 开发者能够在 src/main/org/jboss/chap2/xmbean 目录获得相应的源代码。

列表 2-20 JNDIMap XMBean 版本 1 的描述符

```
<?xml version='1.0' encoding='UTF-8' ?>

<!DOCTYPE server PUBLIC
    "-//JBoss//DTD MBean Service 3.2//EN"
    "http://www.jboss.org/j2ee/dtd/jboss-service_3_2.dtd"
>

<server>
    <mbean code="org.jboss.chap2.xmbean.JNDIMap" name="chap2.xmbean:service=JNDIMap"
        xmbean-dd="META-INF/jndimap-xmbean.xml">
        <attribute name="JndiName">inmemory/maps/MapTest</attribute>
        <depends>jboss:service=Naming</depends>
    </mbean>
</server>

---

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mbean PUBLIC
    "-//JBoss//DTD JBOSS XMBean 1.0//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_xmbean_1_0.dtd">

<mbean>
    <description>The JNDIMap XMBean Example Version 1</description>

    <descriptors>
        <persistence persistPolicy="Never"
            persistPeriod="10"
            persistLocation="data/JNDIMap.data"
            persistName="JNDIMap" />
        <currencyTimeLimit value="10"/>
        <state-action-on-update value="keep-running"/>
    </descriptors>
```



```
<class>org.jboss.test.jmx.xmlbean.JNDIMap</class>

<constructor>
  <description>The default constructor</description>
  <name>JNDIMap</name>
</constructor>

<!-- Attributes -->
<attribute access="read-write" getMethod="getJndiName" setMethod="setJndiName">
  <description>The location in JNDI where the Map we manage will be bound</description>
  <name>JndiName</name>
  <type>java.lang.String</type>
  <descriptors>
    <default value="inmemory/maps/MapTest" />
  </descriptors>
</attribute>
<attribute access="read-write" getMethod="getInitialValues" setMethod="setInitialValues">
  <description>The array of initial values that will be placed into
  the map associated with the service. The array is a collection of
  key,value pairs with elements[0,2,4,...2n] being the keys and
  elements [1,3,5,...,2n+1] the associated values</description>
  <name>InitialValues</name>
  <type>[Ljava.lang.String;</type>
  <descriptors>
    <default value="key0,value0" />
  </descriptors>
</attribute>

<!-- Operations -->
<operation>
  <description>The start lifecycle operation</description>
  <name>start</name>
</operation>
<operation>
  <description>The stop lifecycle operation</description>
  <name>stop</name>
</operation>
<operation impact="ACTION">
  <description>Put a value into the nap</description>
  <name>put</name>
  <parameter>
    <description>The key the value will be store under</description>
    <name>key</name>
```

```

    <type>java.lang.Object</type>
  </parameter>
  <parameter>
    <description>The value to place into the map</description>
    <name>value</name>
    <type>java.lang.Object</type>
  </parameter>
</operation>
<operation impact="INFO">
  <description>Get a value from the map</description>
  <name>get</name>
  <parameter>
    <description>The key to lookup in the map</description>
    <name>get</name>
    <type>java.lang.Object</type>
  </parameter>
  <return-type>java.lang.Object</return-type>
</operation>

<!-- Notifications -->
<notification>
  <description>The notification sent whenever a value is get into the map
    managed by the service</description>
  <name>javax.management.Notification</name>
  <notification-type>org.jboss.chap2.xmlbean.JNDIMap.get</notification-type>
</notification>
<notification>
  <description>The notification sent whenever a value is put into the map
    managed by the service</description>
  <name>javax.management.Notification</name>
  <notification-type>org.jboss.chap2.xmlbean.JNDIMap.put</notification-type>
</notification>
</mbean>

```

正如本文以前提及的一样,JBoss 3.2.2 发布版将绑定的 RMIAdaptor 接口替换成 Invoker 适配器服务。同时,Invoker 适配器服务并不支持远程操作 JMX 通知。因此,这里需要创建并使用 RMIAdaptorService 的配置信息。本书实例提供了 Ant config 目标,用于设置 rmi-adaptor 配置文件集合。其中,该集合安装了随 JBoss 发布的 jmx-rmi-adaptor.sar。具体构建过程如下:

```

[nr@toki]$ ant -Dchap=chap2 config
Buildfile: build.xml

...

```

config:

```
[echo] Preparing rmi-adaptor configuration fileset
[copy] Copying 148 files to /tmp/jboss-3.2.3/server/rmi-adaptor
[copy] Copying 2 files to /tmp/jboss-3.2.3/server/rmi-adaptor/deploy/jmx-rmi-adaptor.sar
[delete] Deleting directory /tmp/jboss-3.2.3/server/rmi-adaptor/deploy/jmx-invoker-adaptor-server.sar
[delete] Deleting directory /tmp/jboss-3.2.3/server/rmi-adaptor/deploy/management
```

其次, 运行 rmi-adaptor 配置。然后, 编译、部署及测试 XMBean。具体如下:

```
[nr@toki examples]$ ant -Dchap=chap2 -Dex=xmbean1 -Djboss.deploy.conf=rmi-adaptor run-example
Buildfile: build.xml
```

validate:

```
[java] ImplementationTitle: JBoss [WonderLand]
[java] ImplementationVendor: JBoss.org
[java] ImplementationVersion: 3.2.2 (build: CVSTag=JBoss_3_2_2 date=200310182216)
[java] SpecificationTitle: JBoss
[java] SpecificationVendor: JBoss (http://www.jboss.org/)
[java] SpecificationVersion: 3.2.2
[java] JBoss version is: 3.2.2
```

fail_if_not_valid:

init:

```
[echo] Using jboss.dist=/cvs/Releases/jboss-3.2.2
```

compile:

run-example:

prepare:

chap2-ex1xmbean1-sar:

run-examplexmbean1:

```
[java] JNDIMap Class: org.jboss.mx.modelmbean.XMBean
[java] JNDIMap Operations:
[java] + void start()
[java] + void stop()
[java] + void put(java.lang.Object chap2.xmbean:service=JNDIMap,java.lang.Object chap2.xmbean:service=JNDIMap)
[java] + java.lang.Object get(java.lang.Object chap2.xmbean:service=JNDIMap)
[java] + java.lang.String getJndiName()
[java] + void setJndiName(java.lang.String chap2.xmbean:service=JNDIMap)
```

```
[java] + [Ljava.lang.String; getInitialValues()
[java] + void setInitialValues([Ljava.lang.String; chap2.xmbean:service=JNDIMap]
[java] handleNotification, event: null
[java] key=key0, value=value0
[java] handleNotification, event: javax.management.Notification[source=chap2.xmbean:service=
JNDIMap,type=org.jboss.chap2.xmbean.JNDIMap.put,sequenceNumber=3,timeStamp=1083610215205,message
=null,userData=null]
[java] JNDIMap.put(key1, value1) successful
[java] handleNotification, event: javax.management.Notification[source=chap2.xmbean:service=
JNDIMap,type=org.jboss.chap2.xmbean.JNDIMap.get,sequenceNumber=4,timeStamp=1083610215376,message
=null,userData=null]
[java] JNDIMap.get(key0): null
[java] handleNotification, event: javax.management.Notification[source=chap2.xmbean:service=
JNDIMap,type=org.jboss.chap2.xmbean.JNDIMap.get,sequenceNumber=5,timeStamp=1083610215406,message
=null,userData=null]
[java] JNDIMap.get(key1): value1
[java] handleNotification, event: javax.management.Notification[source=chap2.xmbean:service=
JNDIMap,type=org.jboss.chap2.xmbean.JNDIMap.put,sequenceNumber=6,timeStamp=1083610215443,message
=null,userData=null]
[java] handleNotification, event: javax.management.AttributeChangeNotification: source=chap2.
xmbean:service=JNDIMap seq-no=3 time=1083610215448 message=InitialValues changed from
javax.management.Attribute: name=InitialValues value=[Ljava.lang.String;@e02ee9 to javax.management.
Attribute: name=InitialValues value=[Ljava.lang.String;@229bc1 attributeName=InitialValues
attributeType=
[Ljava.lang.String;
oldValue=[Ljava.lang.String;@2445d7
newValue=[Ljava.lang.String;@65547d notification Type=jmx.attribute.change userData=null
```

BUILD SUCCESSFUL

Total time: 8 seconds

上述列表演示的功能，除 JMX 通知外，其他功能都和标准 MBean 相同。标准 MBean 不具备发送消息的机制。XMBean 使用 notification 元素声明通知，正如上述展示的 XMBean 版本 1 描述符所示。从测试客户端控制台输出信息可以看到，这些通知来自 get 和 put 操作。有一点请开发者注意，即当 InitialValues 属性发生变动时，也可以发送 jmx.attribute.change 通知。这是 ModelMBean 所具备的标准特性之一。其中，ModelMBean 接口继承于 ModelMBeanNotificationBroadcaster，其支持 AttributeChangeNotificationListener。

对于 JNDIMap 的标准版和 XMBean 版实现而言，其他主要区别在于描述性的元数据。浏览 jmx-console 控制台中的“chap2.xmbean:service=JNDIMap”，开发者将看到如图 2-19 所示的属性列表。

注意

jmx-console 显示的内容是 XMBean 描述符中指定属性的完整描述，而不是标准 MBean 实现中所看到的“MBean Attribute”。将垂直滚动条往下拖到操作部分，开发者将浏览到各操作功能及其参数描述。

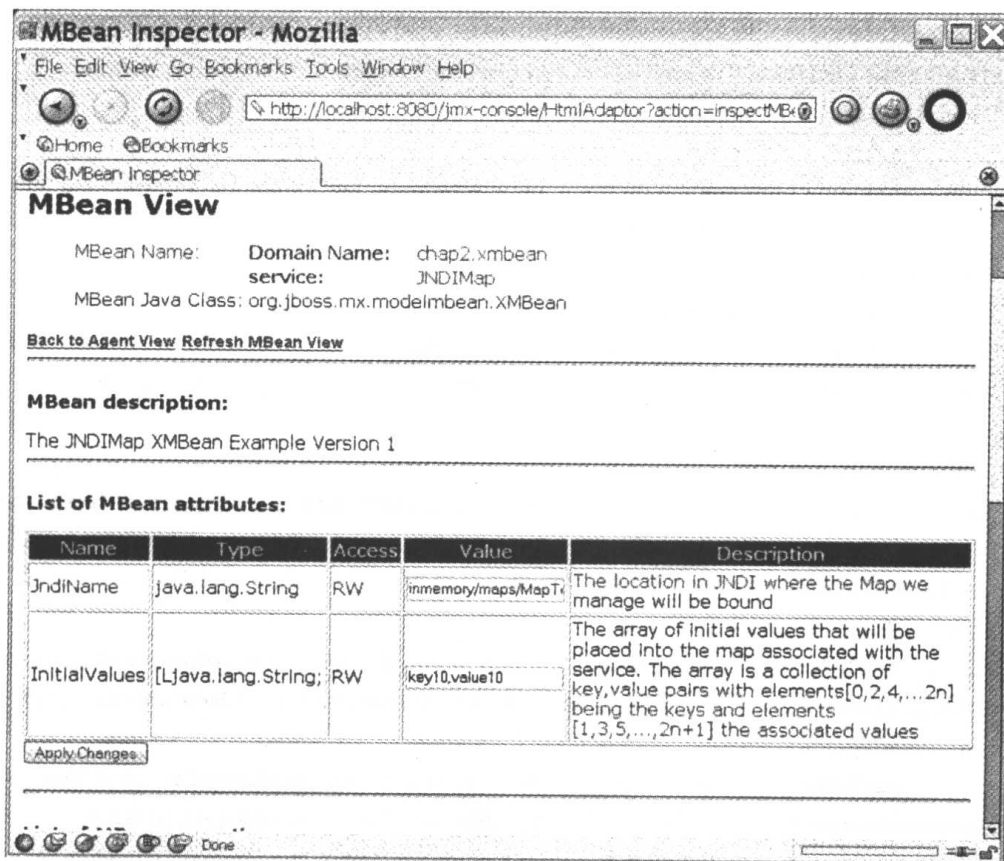


图 2-19 jmx-console 视图中的 JNDIMap XMBean 版本 1

(2) 版本 2, 为 JNDIMap XMBean 添加持久化

XMBean 版本 2 将为 JNDIMap XMBean 属性添加持久化支持。列表 2-21 给出了更新的 XMBean 部署描述符。其中的粗体部分表示其与版本 1 描述符的差异。

列表 2-21 JNDIMap XMBean 描述符版本 2

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mbean PUBLIC
    "-//JBoss//DTD JBOSS XMBean 1.0//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_xmbean_1_0.dtd">

<mbean>
  <description>The JNDIMap XMBean Example Version 2</description>

  <descriptors>
    <persistence persistPolicy="OnUpdate"
      persistPeriod="10"
      persistLocation="{jboss.server.data.dir}"
      persistName="JNDIMap.ser"/>
    <currencyTimeLimit value="10"/>
    <state-action-on-update value="keep-running"/>
    <persistence-manager value="org.jboss.mx.persistence.ObjectStreamPersistenceManager" />
  </descriptors>
</mbean>
```

```

<class>org.jboss.test.jmx.xmbean.JNDIMap</class>

<constructor>
  <description>The default constructor</description>
  <name>JNDIMap</name>
</constructor>

<!-- Attributes -->
<attribute access="read-write" getMethod="getJndiName" setMethod="setJndiName">
  <description>The location in JNDI where the Map we manage will be bound</description>
  <name>JndiName</name>
  <type>java.lang.String</type>
  <descriptors>
    <value value="inmemory/maps/MapTest" />
  </descriptors>
</attribute>
<attribute access="read-write" getMethod="getInitialValues" setMethod="setInitialValues">
  <description>The array of initial values that will be placed into
the map associated with the service. The array is a collection of
key,value pairs with elements[0,2,4,...2n] being the keys and
elements [1,3,5,...,2n+1] the associated values</description>
  <name>InitialValues</name>
  <type>[Ljava.lang.String;</type>
  <descriptors>
    <default value="key0,value0" />
  </descriptors>
</attribute>

<!-- Operations -->
<operation>
  <description>The start lifecycle operation</description>
  <name>start</name>
</operation>
<operation>
  <description>The stop lifecycle operation</description>
  <name>stop</name>
</operation>
<operation impact="ACTION">
  <description>Put a value into the nap</description>
  <name>put</name>
  <parameter>
    <description>The key the value will be store under</description>
    <name>key</name>
    <type>java.lang.Object</type>
  </parameter>

```

```
<parameter>
  <description>The value to place into the map</description>
  <name>value</name>
  <type>java.lang.Object</type>
</parameter>
</operation>
<operation impact="INFO">
  <description>Get a value from the map</description>
  <name>get</name>
  <parameter>
    <description>The key to lookup in the map</description>
    <name>get</name>
    <type>java.lang.Object</type>
  </parameter>
  <return-type>java.lang.Object</return-type>
</operation>

<!-- Notifications -->
<notification>
  <description>The notification sent whenever a value is get into the map
    managed by the service</description>
  <name>javax.management.Notification</name>
  <notification-type>org.jboss.chap2.xmbean.JNDIMap.get</notification-type>
</notification>
<notification>
  <description>The notification sent whenever a value is put into the map
    managed by the service</description>
  <name>javax.management.Notification</name>
  <notification-type>org.jboss.chap2.xmbean.JNDIMap.put</notification-type>
</notification>
</mbean>
```

编译、部署并测试 JNDIMap XMBBean 版本 2 的过程如下：

```
[nr@toki examples]$ ant -Dchap=chap2 -Dex=xmbean2 -Djboss.deploy.conf=rmi-adaptor run-example
Buildfile: build.xml
```

validate:

```
[java] ImplementationTitle: JBoss [WonderLand]
[java] ImplementationVendor: JBoss.org
[java] ImplementationVersion: 3.2.2 (build: CVSTag=JBoss_3_2_2 date=200310182216)
[java] SpecificationTitle: JBoss
[java] SpecificationVendor: JBoss (http://www.jboss.org/)
[java] SpecificationVersion: 3.2.2
[java] JBoss version is: 3.2.2
```

fail_if_not_valid:

```
init:
[echo] Using jboss.dist=/cvs/Releases/jboss-3.2.2

compile:

run-example:

prepare:

chap2-ex1xmbean2-sar:

run-examplexmbean2:
[delete] Deleting: C:\cvs\Releases\jboss-3.2.2\server\rmi-adaptor\deploy\chap2-ex1xmbean1.sar
[copy] Copying 1 file to C:\cvs\Releases\jboss-3.2.2\server\rmi-adaptor\deploy
[java] JNDIMap Class: org.jboss.mx.modelmbean.XMBean
[java] JNDIMap Operations:
[java] + void start()
[java] + void stop()
[java] + void put(java.lang.Object chap2.xmbean:service=JNDIMap,java.lang.Object
chap2.xmbean:service=JNDIMap)
[java] + java.lang.Object get(java.lang.Object chap2.xmbean:service=JNDIMap)
[java] + java.lang.String getJndiName()
[java] + void setJndiName(java.lang.String chap2.xmbean:service=JNDIMap)
[java] + [Ljava.lang.String; getInitialValues()
[java] + void setInitialValues([Ljava.lang.String; chap2.xmbean:service=JNDIMap)
[java] handleNotification, event: null
[java] key=key0, value=value0
[java] handleNotification, event: javax.management.Notification[source=chap2.xmbean:service=J
NDIMap,type=org.jboss.chap2.xmbean.JNDIMap.put,sequenceNumber=3,timeStamp=1068684154664,message=
null,userData=null]
[java] JNDIMap.put(key1, value1) successful
[java] handleNotification, event: javax.management.Notification[source=chap2.xmbean:service=
JNDIMap,type=org.jboss.chap2.xmbean.JNDIMap.get,sequenceNumber=4,timeStamp=1068684154664,message
=null,userData=null]
[java] JNDIMap.get(key0): null
[java] handleNotification, event: javax.management.Notification[source=chap2.xmbean:service=
JNDIMap,type=org.jboss.chap2.xmbean.JNDIMap.get,sequenceNumber=5,timeStamp=1068684154674,message
=null,userData=null]
[java] JNDIMap.get(key1): value1
[java] handleNotification, event: javax.management.Notification[source=chap2.xmbean:service=
JNDIMap,type=org.jboss.chap2.xmbean.JNDIMap.put,sequenceNumber=6,timeStamp=1068684154724,message
=null,userData=null]
[java] handleNotification, event: javax.management.AttributeChangeNotification: source=
chap2.xmbean:service=JNDIMap seq-no=3 time=1068684154744 message=InitialValues changed from
javax.management.Attribute: name=InitialValues value=[Ljava.lang.String;@867fad to
javax.management.Attribute: name=InitialValues value=[Ljava.lang.String;@e62121 attributeName=
```



```
InitialValues attributeType=[Ljava.lang.String; oldValue=[Ljava.lang.String;@1833eca newValue=[Ljava.lang.String;@18f5824 notificationType=jmx.attribute.change userData=null
```

到目前为止，还未发现这两个 XMBean 版本的区别，因为还未开始测试那些已发生持久化的属性。请开发人员重复运行 `xmbean2a` 多次：

```
[nr@toki examples] ant -Dchap=chap2 -Dex=xmbean2a -Djboss.deploy.conf=rmi-adaptor run-example
...
[copy] Copying 1 file to /tmp/jboss-3.2.3/server/rmi-adaptor/deploy
[java] InitialValues.length=2
[java] key=key10, value=value10

[nr@toki examples] ant -Dchap=2 -Dex=xmbean2a -Djboss.deploy.conf=rmi-adaptor run-example
...
run-examplexmbean2a:
[copy] Copying 1 file to /tmp/jboss-3.2.3/server/rmi-adaptor/deploy
[java] InitialValues.length=4
[java] key=key10, value=value10
[java] key=key2, value=value2

[nr@toki examples] ant -Dchap=chap2 -Dex=xmbean2a -Djboss.deploy.conf=rmi-adaptor run-example
...
run-examplexmbean2a:
[copy] Copying 1 file to /tmp/jboss-3.2.3/server/rmi-adaptor/deploy
[java] InitialValues.length=6
[java] key=key10, value=value10
[java] key=key2, value=value2
[java] key=key3, value=value3
```

在运行实例的过程中，这里使用的 `org.jboss.chap2.xmbean.TestXMBeanRestart` 实例能够获得 `InitialValues` 属性的当前取值。然后，它将其他的键/值对添加进来。列表 2-22 给出了客户端代码。

列表 2-22 TestXMBeanRestart 持久化测试客户端

```
package org.jboss.chap2.xmbean;

import javax.management.Attribute;
import javax.management.ObjectName;
import javax.naming.InitialContext;

import org.jboss.jmx.adaptor.rmi.RMIAdaptor;

/** A client that demonstrates the persistence of the xmbean attributes. Every
time it is run it looks up the InitialValues attribute, prints it out
and then adds a new key/value to the list.

@author Scott.Stark@jboss.org
```

```

@version $Revision: 1.2 $
*/
public class TestXMBeanRestart
{
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws Exception
    {
        InitialContext ic = new InitialContext();
        RMIAaptor server = (RMIAaptor) ic.lookup("jmx/rmi/RMIAaptor");

        // Get the InitialValues attribute
        ObjectName name = new ObjectName("chap2.xmbean:service=JNDIMap");
        String[] initialValues = (String[]) server.getAttribute(name, "InitialValues");
        System.out.println("InitialValues.length="+initialValues.length);
        int length = initialValues.length;
        for(int n = 0; n < length; n += 2)
        {
            String key = initialValues[n];
            String value = initialValues[n+1];
            System.out.println("key="+key+", value="+value);
        }
        // Add a new key/value pair
        String[] newInitialValues = new String[length+2];
        System.arraycopy(initialValues, 0, newInitialValues, 0, length);
        newInitialValues[length] = "key"+(length/2+1);
        newInitialValues[length+1] = "value"+(length/2+1);

        Attribute ivalues = new Attribute("InitialValues", newInitialValues);
        server.setAttribute(name, ivalues);
    }
}

```

此时，开发者甚至可以关闭 JBoss 服务器，然后重启它。开发者便能够再次运行初始的命令，以判断持久化操作能否不受服务器重启的影响。

```

[nr@toki examples]$ ant -Dchap=chap2 -Dex=xmbean2 -Djboss.deploy.conf=rmi-adaptor run-example
Buildfile: build.xml
...
chap2-ex1xmbean2-sar:

run-examplexmbean2:
[java] JNDIMap Class: org.jboss.mx.modelImbean.XMBean
[java] JNDIMap Operations:
[java] + void start()
[java] + void stop()

```

```
[java] + void put(java.lang.Object chap2.xmlbean:service=JNDIMap,java.lang.Object chap2.xmlbean:
service=JNDIMap)
[java] + java.lang.Object get(java.lang.Object chap2.xmlbean:service=JNDIMap)
[java] + java.lang.String getJndiName()
[java] + void setJndiName(java.lang.String chap2.xmlbean:service=JNDIMap)
[java] + [Ljava.lang.String; getInitialValues()
[java] + void setInitialValues([Ljava.lang.String; chap2.xmlbean:service=JNDIMap)
[java] handleNotification, event: null
[java] key=key10, value=value10
[java] key=key2, value=value2
[java] key=key3, value=value3
[java] key=key4, value=value4
[java] handleNotification, event: javax.management.Notification[source=chap2.xmlbean:service=
JNDIMap,type=org.jboss.chap2.xmlbean.JNDIMap.put,sequenceNumber=3,timeStamp=1068685268886,message
=null,userData=null]
[java] JNDIMap.put(key1, value1) successful
[java] handleNotification, event: javax.management.Notification[source=chap2.xmlbean:service=
JNDIMap,type=org.jboss.chap2.xmlbean.JNDIMap.get,sequenceNumber=4,timeStamp=1068685268906,message
=null,userData=null]
[java] JNDIMap.get(key0): null
[java] handleNotification, event: javax.management.Notification[source=chap2.xmlbean:service=
JNDIMap,type=org.jboss.chap2.xmlbean.JNDIMap.get,sequenceNumber=5,timeStamp=1068685268906,message
=null,userData=null]
[java] JNDIMap.get(key1): value1
[java] handleNotification, event: javax.management.Notification[source=chap2.xmlbean:service=
JNDIMap,type=org.jboss.chap2.xmlbean.JNDIMap.put,sequenceNumber=6,timeStamp=1068685268976,message
=null,userData=null]
[java] handleNotification, event: javax.management.Notification[source=chap2.xmlbean:service=
JNDIMap,type=org.jboss.chap2.xmlbean.JNDIMap.put,sequenceNumber=7,timeStamp=1068685269006,message
=null,userData=null]
[java] handleNotification, event: javax.management.Notification[source=chap2.xmlbean:service=
JNDIMap,type=org.jboss.chap2.xmlbean.JNDIMap.put,sequenceNumber=8,timeStamp=1068685269016,message
=null,userData=null]
[java] handleNotification, event: javax.management.Notification[source=chap2.xmlbean:service=
JNDIMap,type=org.jboss.chap2.xmlbean.JNDIMap.put,sequenceNumber=9,timeStamp=1068685269016,message
=null,userData=null]
[java] handleNotification, event: javax.management.AttributeChangeNotification: source=
chap2.xmlbean:service=JNDIMap seq-no=5 time=1068685269026 message=InitialValues changed from
javax.management.Attribute: name=InitialValues value=[Ljava.lang.String;@13c6641 to javax.
management.Attribute:name=InitialValues value=[Ljava.lang.String;@f2f0d0 attributeName=
InitialValues attributeType=[Ljava.lang.String; oldValue=[Ljava.lang.String;@145f0e3 newValue=
[Ljava.lang.String;@c9d92c notificationType=jmx.attribute.change userData=null

BUILD SUCCESSFUL
Total time: 8 seconds
```

正如预期的一样，上述对 InitialValues 属性的最后一次设置实际上是可见的。

(3) 版本 3, 为 JNDIMap XMBean 添加安全性和远程访问

JNDIMap XMBean 版本 3 实例将演示服务器拦截器栈的自定义。其中, 本文还将借助于定制的类型化代理将部分 XMBean 管理接口暴露给基于 RMI/JRMP 的远程客户。在服务器端, 将增加一个简单的安全性拦截器, 使得只有在拦截器配置中指定的用户才能够有权限访问属性或操作。与此同时, 还将使用其他自定义拦截器实现 MBean 分离式 Invoker 模式(“2.7 远程访问服务——分离式 Invoker”一节有相关描述)。本文以 Invoker 方式, 而不是借助于 XMBean 实现该模式, 来演示如何在不修改现有 JNDIMap 实现的前提下实现对它的远程访问。

这里将使用 JRMPProxyFactory 服务将列表 2-23 给出的 ClientInterface 暴露给远程客户。这里的测试客户应用将从 JNDI 获得 ClientInterface 代理, 并通过 RMI 方式的调用与 XMBean 进行交互, 而不是通过前面阐述的 RMIAdaptor 和 MBeanServer 方式。见列表 2-24 和列表 2-25。

列表 2-23 借助于 RMI/JRMP 暴露 JNDIMap XMBean 的 ClientInterface 视图

```
public interface ClientInterface
{
    public String[] getInitialValues();
    public void setInitialValues(String[] keyValuePairs);
    public Object get(Object key);
    public void put(Object key, Object value);
}
```

列表 2-24 XMBean 版本 3 的测试客户端

```
package org.jboss.chap2.xmbean;

import javax.naming.InitialContext;
import org.jboss.security.SecurityAssociation;
import org.jboss.security.SimplePrincipal;

/** A client that accesses an XMBean through its RMI interface
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.1 $
 */
public class TestXMBean3
{
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws Exception
    {
        InitialContext ic = new InitialContext();
        ClientInterface xmbean = (ClientInterface) ic.lookup("secure-xmbean/ClientInterface");
    }
}
```

```
// This call should fail because we have not set a security context
try
{
    String[] tmp = xmbean.getInitialValues();
    throw new IllegalStateException("Was able to call getInitialValues");
}
catch(Exception e)
{
    System.out.println("Called to getInitialValues failed as expected: "
        + e.getMessage());
}

// Set a security context using the SecurityAssociation
SecurityAssociation.setPrincipal(new SimplePrincipal("admin"));

// Get the InitialValues attribute
String[] initialValues = xmbean.getInitialValues();
for(int n = 0; n < initialValues.length; n += 2)
{
    String key = initialValues[n];
    String value = initialValues[n+1];
    System.out.println("key="+key+", value="+value);
}

// Invoke the put(Object, Object) op
xmbean.put("key1", "value1");
System.out.println("JNDIMap.put(key1, value1) successful");
Object result0 = xmbean.get("key0");
System.out.println("JNDIMap.get(key0): "+result0);
Object result1 = xmbean.get("key1");
System.out.println("JNDIMap.get(key1): "+result1);

// Change the InitialValues
initialValues[0] += ".1";
initialValues[1] += ".2";
xmbean.setInitialValues(initialValues);

initialValues = xmbean.getInitialValues();
for(int n = 0; n < initialValues.length; n += 2)
{
    String key = initialValues[n];
    String value = initialValues[n+1];
    System.out.println("key="+key+", value="+value);
}
}
```


}

列表 2-25 XMBEAN 版本 3 描述符

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mbean PUBLIC
    "-//JBoss//DTD JBOSS XMBEAN 1.0//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_xmbean_1_0.dtd"
[
    <!-- ATTLIST interceptor adminName CDATA #IMPLIED -->
]
>

<mbean>
    <description>The JNDIMap XMBEAN Example Version 3</description>

    <descriptors>
        <interceptors>
            <interceptor code="org.jboss.chap2.xmbean.ServerSecurityInterceptor"
                adminName="admin"/>
            <interceptor code="org.jboss.chap2.xmbean.InvokerInterceptor" />
            <interceptor code="org.jboss.mx.interceptor.PersistenceInterceptor2" />
            <interceptor code="org.jboss.mx.interceptor.ModelMBeanInterceptor" />
            <interceptor code="org.jboss.mx.interceptor.ObjectReferenceInterceptor" />
        </interceptors>
        <persistence persistPolicy="Never" />
        <currencyTimeLimit value="10"/>
        <state-action-on-update value="keep-running"/>
    </descriptors>

    <class>org.jboss.test.jmx.xmbean.JNDIMap</class>

    <constructor>
        <description>The default constructor</description>
        <name>JNDIMap</name>
    </constructor>

    <!-- Attributes -->
    <attribute access="read-write" getMethod="getJndiName" setMethod="setJndiName">
        <description>The location in JNDI where the Map we manage will be bound</description>
        <name>JndiName</name>
        <type>java.lang.String</type>
        <descriptors>
            <value value="inmemory/maps/MapTest" />
        </descriptors>
    </attribute>

```

```
<attribute access="read-write" getMethod="getInitialValues" setMethod="setInitialValues">
  <description>The array of initial values that will be placed into
  the map associated with the service. The array is a collection of
  key,value pairs with elements[0,2,4,...2n] being the keys and
  elements [1,3,5,...,2n+1] the associated values</description>
  <name>InitialValues</name>
  <type>[Ljava.lang.String;</type>
  <descriptors>
    <value value="key0,value0" />
  </descriptors>
</attribute>

<!-- Operations -->
<operation>
  <description>The start lifecycle operation</description>
  <name>start</name>
</operation>
<operation>
  <description>The stop lifecycle operation</description>
  <name>stop</name>
</operation>
<operation impact="ACTION">
  <description>Put a value into the nap</description>
  <name>put</name>
  <parameter>
    <description>The key the value will be store under</description>
    <name>key</name>
    <type>java.lang.Object</type>
  </parameter>
  <parameter>
    <description>The value to place into the map</description>
    <name>value</name>
    <type>java.lang.Object</type>
  </parameter>
</operation>
<operation impact="INFO">
  <description>Get a value from the map</description>
  <name>get</name>
  <parameter>
    <description>The key to lookup in the map</description>
    <name>get</name>
    <type>java.lang.Object</type>
  </parameter>
  <return-type>java.lang.Object</return-type>
</operation>
```

```

<!-- Notifications -->
<notification>
  <description>The notification sent whenever a value is get into the map
    managed by the service</description>
  <name>javax.management.Notification</name>
  <notification-type>org.jboss.chap2.xmlbean.JNDIMap.get</notification-type>
</notification>
<notification>
  <description>The notification sent whenever a value is put into the map
    managed by the service</description>
  <name>javax.management.Notification</name>
  <notification-type>org.jboss.chap2.xmlbean.JNDIMap.put</notification-type>
</notification>
</mbean>

```

上述列表中粗体部分为 JNDIMap XMBEAN 版本 3 新增的 interceptors 元素。其定义了拦截器栈，即通过这些拦截器访问那些 MBean 的所有属性和操作。最前面的两个拦截器，即 org.jboss.chap2.xmlbean.ServerSecurityInterceptor 和 org.jboss.chap2.xmlbean.InvokerInterceptor，是本实例自定义的拦截器。剩下的 3 个是标准的 ModelMBean 拦截器。由于实例将持久化策略设置为 Never，因此实际上可以将标准的 org.jboss.mx.interceptor.PersistenceInterceptor2 删除。实际上，JMX 拦截器是过滤器链的有序排列。列表 2-26 给出了拦截器的标准基类。

列表 2-26 JMX AbstractInterceptor 基类

```

package org.jboss.mx.interceptor;

import javax.management.MBeanInfo;

import org.jboss.mx.server.MBeanInvoker;

/**
 * Base class for all interceptors.
 *
 * @see org.jboss.mx.interceptor.StandardMBeanInterceptor
 * @see org.jboss.mx.interceptor.LogInterceptor
 *
 * @author <a href="mailto:juha@jboss.org">Juha Lindfors</a>.
 * @version $Revision: 1.4.2.2 $
 */
public class AbstractInterceptor implements Interceptor
{
    // Attributes -----
    protected Interceptor next = null;
    protected String name = null;
    protected MBeanInfo info;

```

```
protected MBeanInvoker invoker;

// Constructors -----
public AbstractInterceptor()
{
    this(null);
}
public AbstractInterceptor(String name)
{
    this.name = name;
}
public AbstractInterceptor(MBeanInfo info, MBeanInvoker invoker)
{
    this.name = getClass().getName();
    this.info = info;
    this.invoker = invoker;
}

// Public -----
public Object invoke(Invocation invocation) throws InvocationException
{
    return getNext().invoke(invocation);
}

public Interceptor getNext()
{
    return next;
}

public Interceptor setNext(Interceptor interceptor)
{
    this.next = interceptor;
    return interceptor;
}
}
```

列表 2-27 和列表 2-28 给出了 XMBEAN 版本 3 实例中的自定义拦截器。其中，列表 2-27 给出了 SecurityInterceptor，列表 2-28 给出了 InvokerInterceptor。

列表 2-27 SecurityInterceptor 拦截器

```
package org.jboss.chap2.xmbean;

import java.security.Principal;

import org.jboss.logging.Logger;
```

```
import org.jboss.mx.interceptor.AbstractInterceptor;
import org.jboss.mx.interceptor.Invocation;
import org.jboss.mx.interceptor.InvocationException;
import org.jboss.security.SimplePrincipal;

/** A simple security interceptor example that restricts access to a single
 * principal
 *
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.1 $
 */
public class ServerSecurityInterceptor extends AbstractInterceptor
{
    private static Logger log = Logger.getLogger(ServerSecurityInterceptor.class);
    private SimplePrincipal admin = new SimplePrincipal("admin");

    public String getAdminName()
    {
        return admin.getName();
    }

    public void setAdminName(String name)
    {
        admin = new SimplePrincipal(name);
    }

    public Object invoke(Invocation invocation) throws InvocationException
    {
        String opName = invocation.getName();
        // If this is not the invoke(Invocation) op just pass it along
        if( opName.equals("invoke") == false )
            return getNext().invoke(invocation);

        Object[] args = invocation.getArgs();
        org.jboss.invocation.Invocation invokeInfo =
            (org.jboss.invocation.Invocation) args[0];
        Principal caller = invokeInfo.getPrincipal();
        log.info("invoke, opName="+opName+", caller="+caller);
        // Only the admin caller is allowed access
        if( caller == null || caller.equals(admin) == false )
        {
            throw new InvocationException(
                new SecurityException("Caller="+caller+" is not allowed access")
            );
        }
        return getNext().invoke(invocation);
    }
}
```


列表 2-28 InvokerInterceptor 拦截器

```
package org.jboss.chap2.xmlbean;

import java.lang.reflect.Method;
import java.util.HashMap;
import javax.management.Descriptor;
import javax.management.MBeanInfo;

import org.jboss.logging.Logger;
import org.jboss.mx.interceptor.AbstractInterceptor;
import org.jboss.mx.interceptor.Invocation;
import org.jboss.mx.interceptor.InvocationException;
import org.jboss.mx.server.MBeanInvoker;
import org.jboss.invocation.MarshalledInvocation;

/** An interceptor that handles the
 *
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.1 $
 */
public class InvokerInterceptor extends AbstractInterceptor
{
    private static Logger log = Logger.getLogger(InvokerInterceptor.class);
    private Class exposedInterface = ClientInterface.class;
    private HashMap methodMap = new HashMap();
    private HashMap invokeMap = new HashMap();

    public InvokerInterceptor(MBeanInfo info, MBeanInvoker invoker)
    {
        super(info, invoker);
        try
        {
            Descriptor[] descriptors = invoker.getDescriptors();
            Object resource = invoker.getResource();
            Class[] getInitialValuesSig = {};
            Method getInitialValues = exposedInterface.getDeclaredMethod("getInitialValues",
                getInitialValuesSig);
            Long hash = new Long(MarshalledInvocation.calculateHash(getInitialValues));
            InvocationInfo invokeInfo = new InvocationInfo("InitialValues",
                Invocation.ATTRIBUTE, Invocation.READ, getInitialValuesSig,
                descriptors, resource);
            methodMap.put(hash, getInitialValues);
            invokeMap.put(getInitialValues, invokeInfo);
        }
    }
}
```

```

        log.debug("getInitialValues hash:"+hash);

        Class[] setInitialValuesSig = {String[].class};
        Method setInitialValues = exposedInterface.getDeclaredMethod("setInitialValues",
            setInitialValuesSig);
        hash = new Long(MarshalledInvocation.calculateHash(setInitialValues));
        invokeInfo = new InvocationInfo("InitialValues",
            Invocation.ATTRIBUTE, Invocation.WRITE, setInitialValuesSig,
            descriptors, resource);
        methodMap.put(hash, setInitialValues);
        invokeMap.put(setInitialValues, invokeInfo);
        log.debug("setInitialValues hash:"+hash);

        Class[] getSig = {Object.class};
        Method get = exposedInterface.getDeclaredMethod("get",
            getSig);
        hash = new Long(MarshalledInvocation.calculateHash(get));
        invokeInfo = new InvocationInfo("get",
            Invocation.OPERATION, Invocation.READ, getSig,
            descriptors, resource);
        methodMap.put(hash, get);
        invokeMap.put(get, invokeInfo);
        log.debug("get hash:"+hash);

        Class[] putSig = {Object.class, Object.class};
        Method put = exposedInterface.getDeclaredMethod("put",
            putSig);
        hash = new Long(MarshalledInvocation.calculateHash(put));
        invokeInfo = new InvocationInfo("put",
            Invocation.OPERATION, Invocation.WRITE, putSig,
            descriptors, resource);
        methodMap.put(hash, put);
        invokeMap.put(put, invokeInfo);
        log.debug("put hash:"+hash);
    }
    catch(Exception e)
    {
        log.error("Failed to init InvokerInterceptor", e);
    }
}

public Object invoke(Invocation invocation) throws InvocationException
{
    String opName = invocation.getName();
    Object[] args = invocation.getArgs();
    Object returnValue = null;

```

```
if( opName.equals("invoke") == true )
{
    org.jboss.invocation.Invocation invokeInfo =
        (org.jboss.invocation.Invocation) args[0];
    // Set the method hash to Method mapping
    if (invokeInfo instanceof MarshalledInvocation)
    {
        MarshalledInvocation mi = (MarshalledInvocation) invokeInfo;
        mi.setMethodMap(methodMap);
    }

    // Invoke the exposedInterface method via reflection if this is an invoke
    Method method = invokeInfo.getMethod();
    Object[] methodArgs = invokeInfo.getArguments();
    InvocationInfo info = (InvocationInfo) invokeMap.get(method);
    Invocation methodInvocation = info.getInvocation(methodArgs);
    returnValue = getNext().invoke(methodInvocation);
}
else
{
    returnValue = getNext().invoke(invocation);
}
return returnValue;
}

/** A class that holds the ClientInterface method info needed to build
 * the JMX Invocation to pass down the interceptor stack.
 */
private class InvocationInfo
{
    private int type;
    private int impact;
    private String name;
    private String[] signature;
    private Descriptor[] descriptors;
    private Object resource;
    InvocationInfo(String name, int type, int impact,
        Class[] signature, Descriptor[] descriptors, Object resource)
    {
        this.name = name;
        this.type = type;
        this.impact = impact;
        this.descriptors = descriptors;
        this.resource = resource;
        this.signature = new String[signature.length];
        for(int s = 0; s < signature.length; s ++)
```

```

    {
        this.signature[s] = signature[s].getName();
    }
}

Invocation getInvocation(Object[] args)
{
    return new Invocation(name, type, impact, args, signature,
        descriptors, resource);
}
}
}

```

其中, `ServerSecurityInterceptor` 拦截 `invoke` 操作, 并验证 `Invocation` 上下文, 以判断其 `principal` 是否为 “admin”。其服务部署描述符见列表 2-29。

`InvokerInterceptor` 实现了分离式 `invoker` 模式。“2.7 远程访问服务——分离式 `Invoker`”一节有详细阐述。

列表 2-29 XMBEAN 版本 3 的服务部署描述符

```

<?xml version='1.0' encoding='UTF-8' ?>
<!--DOCTYPE server
PUBLIC "-//JBoss//DTD MBean Service 3.2//EN"
"http://www.jboss.org/j2ee/dtd/jboss-service_3_2.dtd"

This instance goes beyond the jboss-service_3_2.dtd model
due to its use of the embedded <interceptors> element in the
ClientInterceptors attribute of the JRMPProxyFactory config.
-->

<server>
  <mbean code="org.jboss.chap2.xmbean.JNDIMap"
    name="chap2.xmbean:service=JNDIMap,version=3"
    xmbean-dd="META-INF/jndimap-xmbean3.xml">
    <depends>jboss:service=Naming</depends>
  </mbean>

  <!-- The JRMP invoker proxy configuration for the naming service -->
  <mbean code="org.jboss.invocation.jrmp.server.JRMPProxyFactory"
    name="jboss.test:service=proxyFactory,type=jrmp,target=JNDIMap">
    <!-- Use the standard JRMPInvoker from conf/jboss-service.xml -->
    <attribute name="InvokerName">jboss:service=invoker,type=jrmp</attribute>
    <attribute name="TargetName">chap2.xmbean:service=JNDIMap,version=3</attribute>
    <attribute name="JndiName">secure-xmbean/ClientInterface</attribute>
    <attribute name="ExportedInterface">org.jboss.chap2.xmbean.ClientInterface</attribute>
    <attribute name="ClientInterceptors">
      <interceptors>
        <interceptor>org.jboss.proxy.ClientMethodInterceptor</interceptor>

```

```
<interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
<interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
</interceptors>
</attribute>
<depends>jboss:service=invoker,type=jmmp</depends>
<depends>chap2.xmbean:service=JNDIMap,version=3</depends>
</mbean>

</server>

[nr@toki examples] ant -Dchap=chap2 -Dex=xmbean3 config
...
config:
[echo] Preparing rmi-adaptor configuration fileset
[copy] Copying 5 files to /tmp/jboss-3.2.3/server/rmi-adaptor
[delete] Deleting directory /tmp/jboss-3.2.3/server/rmi-adaptor/deploy/jmx-invoker-adaptor-server.sar
[delete] Deleting directory /tmp/jboss-3.2.3/server/rmi-adaptor/deploy/management

[nr@toki examples]$ ant -Dchap=chap2 -Dex=xmbean3 run-example
...
run-examplexmbean3:
[java] Called to getInitialValues failed as expected: Caller=null is not allowed access
[java] key=key0, value=value0
[java] JNDIMap.put(key1, value1) successful
[java] JNDIMap.get(key0): null
[java] JNDIMap.get(key1): value1
[java] key=key0.1, value=value0.2

[starksm@banshee examples]$ ant -Dchap=chap2 -Dex=xmbean3 run-example
...
run-examplexmbean3:
[java] Called to getInitialValues failed as expected: Caller=null is not allowed access
[java] key=key0.1, value=value0.2
[java] JNDIMap.put(key1, value1) successful
[java] JNDIMap.get(key0): null
[java] JNDIMap.get(key1): value1
[java] key=key0.1.1, value=value0.2.2
```

2.4.4 部署排序和依赖性

本文已经阐述了如何通过服务描述符中的 `depends` 和 `depends-list` 标签管理依赖性。由于存在一种顺序以完成部署单元的部署工作，因此部署单元扫描器支持的部署排序提供了粗粒度的依赖性管理。如果依赖性和部署包能够保持一致，则能够有更简单的机制实现依赖性管理，而不用显式地列举出 `MBean` 与 `MBean` 之间的依赖关系。通过开发自定义的过滤器，开发者能够改动部署扫描器完成的粗粒度排序工作。

在完成某组件存档的部署后, JBoss 是通过深度优先排序处理其嵌入的子部署单元的部署任务的。另外, 开发者还可以将组件结构化存储到具有层次结构的存档中。

通常情况下, 由于包结构并不会解析依赖性, 因此需要显式地给出 MBean 的依赖性。接下来考虑某组件部署单元实例, 其由 MBean 及其使用的 EJB 组成。列表 2-30 给出了实例 EAR 的结构。

列表 2-30 某含有 MBean 及其依赖的 EJB 的 EAR 实例

```
output/chap2/chap2-ex3.ear
+- META-INF/MANIFEST.MF
+- META-INF/jboss-app.xml
+- chap2-ex3.jar (archive) [EJB jar]
| +- META-INF/MANIFEST.MF
| +- META-INF/ejb-jar.xml
| +- org/jboss/chap2/ex3/EchoBean.class
| +- org/jboss/chap2/ex3/EchoLocal.class
| +- org/jboss/chap2/ex3/EchoLocalHome.class
+- chap2-ex3.sar (archive) [MBean sar]
| +- META-INF/MANIFEST.MF
| +- META-INF/jboss-service.xml
| +- org/jboss/chap2/ex3/EjbMBeanAdaptor.class
+- META-INF/application.xml
```

该 EAR 含有 chap2-ex3.jar 和 chap2-ex3.sar。chap2-ex3.jar 为 EJB 存档, chap2-ex3.sar 为 MBean 服务存档。这里以 DynamicMBean 形式实现了该服务, 并给出了其使用方法。列表 2-31 给出了用于 EjbMBeanAdaptor MBean 服务的源代码。

列表 2-31 使用 EJB 的、基于 DynamicMBean 的 MBean 服务

```
301 package org.jboss.chap2.ex3;
302
303 import java.lang.reflect.Method;
304 import javax.ejb.CreateException;
305 import javax.management.Attribute;
306 import javax.management.AttributeList;
307 import javax.management.AttributeNotFoundException;
308 import javax.management.DynamicMBean;
309 import javax.management.InvalidAttributeValueException;
310 import javax.management.JMRuntimeException;
311 import javax.management.MBeanAttributeInfo;
312 import javax.management.MBeanConstructorInfo;
313 import javax.management.MBeanInfo;
314 import javax.management.MBeanNotificationInfo;
315 import javax.management.MBeanOperationInfo;
316 import javax.management.MBeanException;
317 import javax.management.MBeanServer;
318 import javax.management.ObjectName;
```

```
319 import javax.management.ReflectionException;
320 import javax.naming.InitialContext;
321 import javax.naming.NamingException;
322
323 import org.jboss.system.ServiceMBeanSupport;
324
325 /** An example of a DynamicMBean that exposes select attributes and operations
326 of an EJB as an MBean.
327
328 @author Scott.Stark@jboss.org
329 @version $Revision: 1.1 $
330 */
331 public class EjbMBeanAdaptor extends ServiceMBeanSupport
332 implements DynamicMBean
333 {
334 private String helloPrefix;
335 private String ejbJndiName;
336 private EchoLocalHome home;
337
338 /** These are the mbean attributes we expose
339 */
340 private MBeanAttributeInfo[] attributes = {
341     new MBeanAttributeInfo("HelloPrefix", "java.lang.String",
342         "The prefix message to append to the session echo reply",
343         true, // isReadable
344         true, // isWritable
345         false), // isIs
346     new MBeanAttributeInfo("EjbJndiName", "java.lang.String",
347         "The JNDI name of the session bean local home",
348         true, // isReadable
349         true, // isWritable
350         false) // isIs
351 };
352 /** These are the mbean operations we expose
353 */
354 private MBeanOperationInfo[] operations;
355
356
357 /** We override this method to setup our echo operation info. It could
358 also be done in a ctor.
359 */
360 public ObjectName preRegister(MBeanServer server, ObjectName name)
361     throws Exception
362 {
363     log.info("preRegister notification seen");
364
```

```
365 operations = new MBeanOperationInfo[5];
366
367 Class thisClass = getClass();
368 Class[] parameterTypes = {String.class};
369 Method echoMethod = thisClass.getMethod("echo", parameterTypes);
370 String desc = "The echo op invokes the session bean echo method and"
371 + " returns its value prefixed with the helloPrefix attribute value";
372 operations[0] = new MBeanOperationInfo(desc, echoMethod);
373
374 // Add the Service interface operations from our super class
375 parameterTypes = new Class[0];
376 Method createMethod = thisClass.getMethod("create", parameterTypes);
377 operations[1] = new MBeanOperationInfo("The JBoss Service.create", createMethod);
378 Method startMethod = thisClass.getMethod("start", parameterTypes);
379 operations[2] = new MBeanOperationInfo("The JBoss Service.start", startMethod);
380 Method stopMethod = thisClass.getMethod("stop", parameterTypes);
381 operations[3] = new MBeanOperationInfo("The JBoss Service.stop", stopMethod);
382 Method destroyMethod = thisClass.getMethod("destroy", parameterTypes);
383 operations[4] = new MBeanOperationInfo("The JBoss Service.destroy", destroyMethod);
384 return name;
385 }
386
387
388 // --- Begin ServiceMBeanSupport overrides
389 protected void createService() throws Exception
390 {
391     log.info("Notified of create state");
392 }
393 protected void startService() throws Exception
394 {
395     log.info("Notified of start state");
396     InitialContext ctx = new InitialContext();
397     home = (EchoLocalHome) ctx.lookup(ejbIndiName);
398 }
399 protected void stopService()
400 {
401     log.info("Notified of stop state");
402 }
403 // --- End ServiceMBeanSupport overrides
404
405 public String getHelloPrefix()
406 {
407     return helloPrefix;
408 }
409 public void setHelloPrefix(String helloPrefix)
410 {
```

```
411     this.helloPrefix = helloPrefix;
412 }
413
414 public String getEjbJndiName()
415 {
416     return ejbJndiName;
417 }
418 public void setEjbJndiName(String ejbJndiName)
419 {
420     this.ejbJndiName = ejbJndiName;
421 }
422
423 public String echo(String arg)
424     throws CreateException, NamingException
425 {
426     log.debug("Lookup EchoLocalHome@"+ejbJndiName);
427     EchoLocal bean = home.create();
428     String echo = helloPrefix + bean.echo(arg);
429     return echo;
430 }
431
432 // --- Begin DynamicMBean interface methods
433 /** Returns the management interface that describes this dynamic resource.
434  * It is the responsibility of the implementation to make sure the
435  * description is accurate.
436  *
437  * @return the management interface descriptor.
438  */
439 public MBeanInfo getMBeanInfo()
440 {
441     String classname = getClass().getName();
442     String description = "This is an MBean that uses a session bean in the"
443         + " implementation of its echo operation.";
444     MBeanConstructorInfo[] constructors = null;
445     MBeanNotificationInfo[] notifications = null;
446     MBeanInfo mbeanInfo = new MBeanInfo(classname, description, attributes,
447         constructors, operations, notifications);
448     // Log when this is called so we know when in the lifecycle this is used
449     Throwable trace = new Throwable("getMBeanInfo trace");
450     log.info("Don't panic, just a stack trace", trace);
451     return mbeanInfo;
452 }
453
454 /** Returns the value of the attribute with the name matching the
455  * passed string.
456  *
```

```
457 * @param attribute the name of the attribute.
458 * @return the value of the attribute.
459 * @exception AttributeNotFoundException when there is no such attribute.
460 * @exception MBeanException wraps any error thrown by the resource when
461 * getting the attribute.
462 * @exception ReflectionException wraps any error invoking the resource.
463 */
464 public Object getAttribute(String attribute)
465     throws AttributeNotFoundException, MBeanException, ReflectionException
466 {
467     Object value = null;
468     if( attribute.equals("HelloPrefix") )
469         value = getHelloPrefix();
470     else if( attribute.equals("EjbJndiName") )
471         value = getEjbJndiName();
472     else
473         throw new AttributeNotFoundException("Unknown attribute("+attribute+")
requested");
474     return value;
475 }
476
477 /** Returns the values of the attributes with names matching the
478 * passed string array.
479 *
480 * @param attributes the names of the attribute.
481 * @return an {@link AttributeList AttributeList} of name and value pairs.
482 */
483 public AttributeList getAttributes(String[] attributes)
484 {
485     AttributeList values = new AttributeList();
486     for(int a = 0; a < attributes.length; a++)
487     {
488         String name = attributes[a];
489         try
490         {
491             Object value = getAttribute(name);
492             Attribute attr = new Attribute(name, value);
493             values.add(attr);
494         }
495         catch(Exception e)
496         {
497             log.error("Failed to find attribute: "+name, e);
498         }
499     }
500     return values;
501 }
```



```
502
503 /** Sets the value of an attribute. The attribute and new value are
504 * passed in the name value pair {@link Attribute Attribute}.
505 *
506 * @see javax.management.Attribute
507 *
508 * @param attribute the name and new value of the attribute.
509 * @exception AttributeNotFoundException when there is no such attribute.
510 * @exception InvalidAttributeValueException when the new value cannot be
511 * converted to the type of the attribute.
512 * @exception MBeanException wraps any error thrown by the resource when
513 * setting the new value.
514 * @exception ReflectionException wraps any error invoking the resource.
515 */
516 public void setAttribute(Attribute attribute)
517     throws AttributeNotFoundException, InvalidAttributeValueException,
518     MBeanException, ReflectionException
519 {
520     String name = attribute.getName();
521     if( name.equals("HelloPrefix") )
522     {
523         String value = attribute.getValue().toString();
524         setHelloPrefix(value);
525     }
526     else if( name.equals("EjbJndiName") )
527     {
528         String value = attribute.getValue().toString();
529         setEjbJndiName(value);
530     }
531     else
532         throw new AttributeNotFoundException("Unknown attribute("+name+")
requested");
533 }
534
535 /** Sets the values of the attributes passed as an
536 * {@link AttributeList AttributeList} of name and new value pairs.
537 *
538 * @param attributes the name and new value pairs.
539 * @return an {@link AttributeList AttributeList} of name and value pairs
540 * that were actually set.
541 */
542 public AttributeList setAttributes(AttributeList attributes)
543 {
544     AttributeList setAttributes = new AttributeList();
545     for(int a = 0; a < attributes.size(); a++)
546     {
```

```
547     Attribute attr = (Attribute) attributes.get(a);
548     try
549     {
550         setAttribute(attr);
551         setAttributes.add(attr);
552     }
553     catch(Exception ignore)
554     {
555     }
556 }
557 return setAttributes;
558 }
559
560 /** Invokes a resource operation.
561 *
562 * @param actionName the name of the operation to perform.
563 * @param params the parameters to pass to the operation.
564 * @param signature the signartures of the parameters.
565 * @return the result of the operation.
566 * @exception MBeanException wraps any error thrown by the resource when
567 * performing the operation.
568 * @exception ReflectionException wraps any error invoking the resource.
569 */
570 public Object invoke(String actionName, Object[] params, String[] signature)
571     throws MBeanException, ReflectionException
572 {
573     Object rtnValue = null;
574     log.debug("Begin invoke, actionName="+actionName);
575     try
576     {
577         if( actionName.equals("echo") )
578         {
579             String arg = (String) params[0];
580             rtnValue = echo(arg);
581             log.debug("Result: "+rtnValue);
582         }
583         else if( actionName.equals("create") )
584         {
585             super.create();
586         }
587         else if( actionName.equals("start") )
588         {
589             super.start();
590         }
591         else if( actionName.equals("stop") )
592         {
```

```
593         super.stop();
594     }
595     else if( actionName.equals("destroy") )
596     {
597         super.destroy();
598     }
599     else
600     {
601         throw new JMRuntimeException("Invalid state, don't know about
op="+actionName);
602     }
603 }
604 catch(Exception e)
605 {
606     throw new ReflectionException(e, "echo failed");
607 }
608 log.debug("End invoke, actionName="+actionName);
609 return rtnValue;
610 }
611
612 // --- End DynamicMBean interface methods
613
614 }
```

信不信由你，这是一个非常试验性的 MBean。这里的大部分代码用于提供 MBean 元数据，并处理来自 MBeanServer 的回调。这也是 DynamicMBean 所要求的，因为开发者能够随意暴露任何所需的管理接口。事实上，借助于 getMBeanInfo 方法返回不同的元数据值，DynamicMBean 便能够在运行时动态地变更其管理接口。当然，客户端可能对这样的动态对象比较反感，但 MBeanServer 对 DynamicMBean 变更其接口视而不见，即并不阻止它。

该实例展示了两方面的内容。其一，证明了 MBean 如何依赖于 EJB 完成其部分功能。其二，采用动态管理接口创建 MBean。如果需要借助于静态接口开发该实例的标准 MBean 版本，则可以根据列表 2-32 实现。

列表 2-32 为列表 2-31 提供标准 MBean 接口

```
public interface EjbMBeanAdaptorMBean
{
    public String getHelloPrefix();
    public void setHelloPrefix(String prefix);
    public String getEjbJndiName();
    public void setEjbJndiName(String jndiName);
    public String echo(String arg) throws CreateException, NamingException;
    public void create() throws Exception;
```

```
public void start() throws Exception;  
public void stop();  
public void destroy();  
}
```

第 67~83 行用于构建 MBean 操作元数据。其中, `echo(String)`、`create()`、`start()`、`stop()` 及 `destroy()` 操作由各自的 `java.lang.reflect.Method` 对象定义, 同时也添加了各自的描述性说明。接下来, 让我们看看接口实现的具体位置及 MBean 是如何使用 EJB 的。在第 40~51 行创建的两个 `MBeanAttributeInfo` 实例定义了 MBean 的属性。这些属性对应于静态接口的 `getHelloPrefix/setHelloPrefix` 和 `getEjbJndiName/setEjbJndiName` 方法。使用 `DynamicMBean` 的动机之一是能够为属性元数据添加描述性字符串, 而静态接口却无能为力。

第 88~103 行对应于 JBoss 服务生命周期回调。由于本实例继承于 `ServiceMBeanSupport` 实用类, 因此这里重载了 `createService`、`startService` 及 `stopService` 模板回调, 而不是 `Service` 接口中的 `create`、`start` 及 `stop` 方法。有一点请开发者注意, 即在使用 `startService` 方法之前, 还不能够查找到 EJB 的 `EchoLocalHome` 接口。在这之前, 试图访问其 `Home` 接口必然会出现 JNDI 没有被绑定的错误消息, 因为 EJB 容器还没有绑定 `Home` 接口。由于存在依赖关系, 使得在 EJB 容器没有启动时, 依赖于运行 `EchoLocal` 的 EJB 容器的 MBean 服务也不能够启动。本文在浏览服务描述符的时候, 开发者将看到这种依赖约定。

第 105~121 行实现了 `HelloPrefix` 和 `EjbJndiName` 属性的访问操作。借助于 `MBeanServer`, 开发者能够使用 `getAttribute/setAttribute` 调用访问到上述属性。

第 123~130 行实现了 `echo (String)` 操作。该方法调用 `EchoLocal.echo(String)` EJB 方法。通过 `startService` 方法获得的 `EchoLocalHome` 创建了本地 Bean 接口。

该类的其他部分组成了 `DynamicMBean` 接口实现。第 133~152 行对应于 MBean 元数据访问者回调。该方法返回 `javax.management.MBeanInfo` 形式的对象, 即描述 MBean 的管理接口。其中, 它包含了先前创建的 `MBeanAttributeInfo` 和 `MBeanOperationInfo` 元数据、构建器及通知信息。由于该 MBean 三需要额外的构建器或通知, 所以这些信息都为 `null`。

第 154~258 行用于处理属性访问请求。这项工作相当讨厌, 而且容易出错, 因此建议使用工具包, 或基础框架以帮助生成这些方法。基于 XML 的 `ModelMBean` 框架 (称为 `XMBeans`) 当前正由 JBoss 研发。除此之外, 当前并不存在其他的 `DynamicMBean` 框架。

第 260~310 行对应于操作调用的分发入口点。MBean 处理请求中发送的 `actionName`, 并调用合适的方法。

列表 2-33 给出了用于该 MBean 的 `jboss-service.xml` 描述符, 并用粗体凸显 MBean 依赖的 EJB 容器服务。EJB 容器 MBean `ObjectName` 格式为:

```
"jboss.j2ee:service=EJB,jndiName=" + <home-jndi-name>
```

其中, `<home-jndi-name>` 指 EJB `Home` 接口的 JNDI 名字。

列表 2-33 DynamicMBean jboss-service.xml 描述符

```
<server>  
  <mbean code="org.jboss.chap2.ex3.EjbMBeanAdaptor"
```

```

        name="jboss.book:service=EjbMBeanAdaptor">
        <attribute name="HelloPrefix">AdaptorPrefix</attribute>
        <attribute name="EjbJndiName">local/chap2.EchoBean</attribute>
        <depends>jboss.j2ee:service=EJB,jndiName=local/chap2.EchoBean</depends>
    </mbean>
</server>

```

运行如下命令行实现部署目的:

```

(starksm@banshee examples)$ ant -Dchap=chap2 -Dex=3 run-example
...
run-example3:
    [copy] Copying 1 file to C:\cvs\Releases\jboss-3.2.2\server\default\deploy

BUILD SUCCESSFUL
Total time: 1 second

```

服务器控制台将出现如下的类似信息:

```

15:00:38,457 INFO [MainDeployer] Starting deployment of package: file:/private/tmp/jboss-3.2.3/server/
default/deploy/chap2-ex3.ear
15:00:38,564 INFO [EARDeployer] Init J2EE application: file:/private/tmp/jboss-3.2.3/server/default/
deploy/chap2-ex3.ear
15:00:38,875 INFO [EjbMBeanAdaptor] preRegister notification seen
15:00:38,928 INFO [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace at org.jboss.chap2.ex3.EjbMBeanAdaptor.getMBeanInfo
(EjbMBeanAdaptor.java:153)
    at org.jboss.mx.server.MBeanServerImpl.getMBeanInfo(MBeanServerImpl.java:568)
    at org.jboss.system.ServiceConfigurator.configure(ServiceConfigurator.java:215)
    at org.jboss.system.ServiceConfigurator.internalInstall(ServiceConfigurator.java:172)
    at org.jboss.system.ServiceConfigurator.install(ServiceConfigurator.java:114)
    at org.jboss.system.ServiceController.install(ServiceController.java:225)
    at sun.reflect.GeneratedMethodAccessor29.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:324)
    at org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
    at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:546)
    at org.jboss.mx.util.MBeanProxyExt.invoke(MBeanProxyExt.java:177)
    at $Proxy4.install(Unknown Source) at
org.jboss.deployment.SARDeployer.create(SARDeployer.java:183) at org.jboss.deployment.Main
Deployer.create(MainDeployer.java:786)
    at org.jboss.deployment.MainDeployer.create(MainDeployer.java:778)
    at org.jboss.deployment.MainDeployer.deploy(MainDeployer.java:641)
    at org.jboss.deployment.MainDeployer.deploy(MainDeployer.java:605)
    at sun.reflect.GeneratedMethodAccessor20.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:324)

```



```
at org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:546)
at org.jboss.mx.util.MBeanProxyExt.invoke(MBeanProxyExt.java:177)
at $Proxy6.deploy(Unknown Source)
at org.jboss.deployment.scanner.URLDeploymentScanner.deploy(URLDeploymentScanner.java:302)
at org.jboss.deployment.scanner.URLDeploymentScanner.scan(URLDeploymentScanner.java:476)
at org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.doScan(Abstract
DeploymentScanner.java:201)
at org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.loop(Abstract
DeploymentScanner.java:212)
at org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.run(Abstract
DeploymentScanner.java:191)
15:00:39,028 INFO [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
at org.jboss.chap2.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:153)
at org.jboss.mx.server.MBeanServerImpl.getMBeanInfo(MBeanServerImpl.java:568)
at org.jboss.system.ServiceController.getServiceProxy(ServiceController.java:745)
at org.jboss.system.ServiceController.create(ServiceController.java:276)
at org.jboss.system.ServiceController.create(ServiceController.java:243)
at sun.reflect.GeneratedMethodAccessor4.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:324)
at org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:546)
at org.jboss.mx.util.MBeanProxyExt.invoke(MBeanProxyExt.java:177)
at $Proxy4.create(Unknown Source)
at org.jboss.deployment.SARDeployer.create(SARDeployer.java:192)
at org.jboss.deployment.MainDeployer.create(MainDeployer.java:786)
at org.jboss.deployment.MainDeployer.create(MainDeployer.java:778)
at org.jboss.deployment.MainDeployer.deploy(MainDeployer.java:641)
at org.jboss.deployment.MainDeployer.deploy(MainDeployer.java:605)
at sun.reflect.GeneratedMethodAccessor20.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:324)
at org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:546)
at org.jboss.mx.util.MBeanProxyExt.invoke(MBeanProxyExt.java:177)
at $Proxy6.deploy(Unknown Source)
at org.jboss.deployment.scanner.URLDeploymentScanner.deploy(URLDeploymentScanner.java:302)
at org.jboss.deployment.scanner.URLDeploymentScanner.scan(URLDeploymentScanner.java:476)
at org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.doScan(Abstract
DeploymentScanner.java:201)
at org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.loop(Abstract
DeploymentScanner.java:212)
at org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.run(Abstract
DeploymentScanner.java:191)
```

```
15:00:41,075 INFO [EjbModule] Deploying Chap2EchoInfoBean
15:00:41,652 INFO [EjbModule] Deploying chap2.EchoBean
15:00:42,107 INFO [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
    at org.jboss.chap2.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:153)
    at org.jboss.mx.server.MBeanServerImpl.getMBeanInfo(MBeanServerImpl.java:568)
    at org.jboss.system.ServiceController.getServiceProxy(ServiceController.java:745)
    at org.jboss.system.ServiceController.create(ServiceController.java:276)
    at org.jboss.system.ServiceController.create(ServiceController.java:243)
    at org.jboss.system.ServiceController.create(ServiceController.java:333)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:324)
    at org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
    at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:546)
    at org.jboss.mx.util.MBeanProxyExt.invoke(MBeanProxyExt.java:177)
    at $Proxy31.create(Unknown Source)
    at org.jboss.ejb.EjbModule.createService(EjbModule.java:301)
    at org.jboss.system.ServiceMBeanSupport.create(ServiceMBeanSupport.java:158)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:324)
    at org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
    at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:546)
    at org.jboss.system.ServiceController$ServiceProxy.invoke(ServiceController.java:976)
    at $Proxy14.create(Unknown Source)
    at org.jboss.system.ServiceController.create(ServiceController.java:310)
    at org.jboss.system.ServiceController.create(ServiceController.java:243)
    at sun.reflect.GeneratedMethodAccessor4.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:324)
    at org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
    at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:546)
    at org.jboss.mx.util.MBeanProxyExt.invoke(MBeanProxyExt.java:177)
    at $Proxy12.create(Unknown Source)
    at org.jboss.ejb.EJBDeployer.create(EJBDeployer.java:523)
    at org.jboss.deployment.MainDeployer.create(MainDeployer.java:786)
    at org.jboss.deployment.MainDeployer.create(MainDeployer.java:778)
    at org.jboss.deployment.MainDeployer.deploy(MainDeployer.java:641)
    at org.jboss.deployment.MainDeployer.deploy(MainDeployer.java:605)
    at sun.reflect.GeneratedMethodAccessor20.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:324)
    at org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
```

```
at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:546)
at org.jboss.mx.util.MBeanProxyExt.invoke(MBeanProxyExt.java:177)
at $Proxy6.deploy(Unknown Source)
at org.jboss.deployment.scanner.URLDeploymentScanner.deploy(URLDeploymentScanner.java:302)
at org.jboss.deployment.scanner.URLDeploymentScanner.scan(URLDeploymentScanner.java:476)
at org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.doScan(Abstract
DeploymentScanner.java:201)
at org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.loop(Abstract
DeploymentScanner.java:212)
at org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.run(Abstract
DeploymentScanner.java:191)
15:00:42,693 INFO [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
at org.jboss.chap2.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:153)
at org.jboss.mx.server.MBeanServerImpl.getMBeanInfo(MBeanServerImpl.java:568)
at org.jboss.system.ServiceController.getServiceProxy(ServiceController.java:745)
at org.jboss.system.ServiceController.create(ServiceController.java:276)
at org.jboss.system.ServiceController.create(ServiceController.java:243)
at org.jboss.system.ServiceController.create(ServiceController.java:333)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:324)
at org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:546)
at org.jboss.mx.util.MBeanProxyExt.invoke(MBeanProxyExt.java:177)
at $Proxy31.create(Unknown Source)
at org.jboss.ejb.EjbModule.createService(EjbModule.java:301)
at org.jboss.system.ServiceMBeanSupport.create(ServiceMBeanSupport.java:158)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:324)
at org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:546)
at org.jboss.system.ServiceController$ServiceProxy.invoke(ServiceController.java:976)
at $Proxy14.create(Unknown Source)
at org.jboss.system.ServiceController.create(ServiceController.java:310)
at org.jboss.system.ServiceController.create(ServiceController.java:243)
at sun.reflect.GeneratedMethodAccessor4.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:324)
at org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:546)
at org.jboss.mx.util.MBeanProxyExt.invoke(MBeanProxyExt.java:177)
at $Proxy12.create(Unknown Source) at org.jboss.ejb.EJBDeployer.create(EJBDeployer.java:523)
```

```
at org.jboss.deployment.MainDeployer.create(MainDeployer.java:786)
at org.jboss.deployment.MainDeployer.create(MainDeployer.java:778)
at org.jboss.deployment.MainDeployer.deploy(MainDeployer.java:641)
at org.jboss.deployment.MainDeployer.deploy(MainDeployer.java:605)
at sun.reflect.GeneratedMethodAccessor20.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:324)
at org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:546)
at org.jboss.mx.util.MBeanProxyExt.invoke(MBeanProxyExt.java:177)
at $Proxy6.deploy(Unknown Source)
at org.jboss.deployment.scanner.URLDeploymentScanner.deploy(URLDeploymentScanner.java:302)
at org.jboss.deployment.scanner.URLDeploymentScanner.scan(URLDeploymentScanner.java:476)
at org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.doScan(Abstract
DeploymentScanner.java:201)
at org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.loop(Abstract
DeploymentScanner.java:212)
at org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.run(Abstract
DeploymentScanner.java:191)
15:00:42,761 INFO [EjbMBeanAdaptor] Begin invoke, actionName=create
15:00:42,768 INFO [EjbMBeanAdaptor] Notified of create state
15:00:42,769 INFO [EjbMBeanAdaptor] End invoke, actionName=create
15:00:42,780 INFO [EjbMBeanAdaptor] Don't panic, just a stack trace
java.lang.Throwable: getMBeanInfo trace
at org.jboss.chap2.ex3.EjbMBeanAdaptor.getMBeanInfo(EjbMBeanAdaptor.java:153)
at org.jboss.mx.server.MBeanServerImpl.getMBeanInfo(MBeanServerImpl.java:568)
at org.jboss.mx.util.JMXInvocationHandler.<init>(JMXInvocationHandler.java:66)
at org.jboss.mx.util.MBeanProxy.get(MBeanProxy.java:79)
at org.jboss.mx.util.MBeanProxy.get(MBeanProxy.java:67)
at org.jboss.management.j2ee.MBean.postCreation(MBean.java:154)
at org.jboss.management.j2ee.J2EEManagedObject.postRegister(J2EEManagedObject.java:250)
at org.jboss.mx.server.registry.BasicMBeanRegistry.registerMBean(BasicMBeanRegistry.java:278)
at sun.reflect.GeneratedMethodAccessor1.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:324)
at org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
at org.jboss.mx.interceptor.ObjectReferenceInterceptor.invoke(ObjectReferenceInterceptor.java:59)
at org.jboss.mx.interceptor.MBeanAttributeInterceptor.invoke(MBeanAttributeInterceptor.java:43)
at org.jboss.mx.interceptor.PersistenceInterceptor2.invoke(PersistenceInterceptor2.java:93)
at org.jboss.mx.server.MBeanInvoker.invoke(MBeanInvoker.java:76)
at javax.management.modelmbean.RequiredModelMBean.invoke(RequiredModelMBean.java:144)
at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:546)
at org.jboss.mx.server.MBeanServerImpl.registerMBean(MBeanServerImpl.java:997)
at org.jboss.mx.server.MBeanServerImpl.registerMBean(MBeanServerImpl.java:327)
at org.jboss.management.j2ee.MBean.create(MBean.java:65)
at org.jboss.management.j2ee.factory.ServiceModuleFactory.create(ServiceModuleFactory.java:53)
```



```
at org.jboss.management.j2ee.LocalJBossServerDomain.handleNotification(LocalJBossServer
Domain.java:383)
at org.jboss.mx.server.NotificationListenerProxy.handleNotification(NotificationListenerProxy.java:69)
at javax.management.NotificationBroadcasterSupport.sendNotification(Notification
BroadcasterSupport.java:93)
at org.jboss.deployment.SubDeployerSupport.start(SubDeployerSupport.java:178)
at org.jboss.deployment.SARDeployer.start(SARDeployer.java:229)
at org.jboss.deployment.MainDeployer.start(MainDeployer.java:832)
at org.jboss.deployment.MainDeployer.start(MainDeployer.java:824)
at org.jboss.deployment.MainDeployer.deploy(MainDeployer.java:642)
at org.jboss.deployment.MainDeployer.deploy(MainDeployer.java:605)
at sun.reflect.GeneratedMethodAccessor20.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:324)
at org.jboss.mx.capability.ReflectedMBeanDispatcher.invoke(ReflectedMBeanDispatcher.java:284)
at org.jboss.mx.server.MBeanServerImpl.invoke(MBeanServerImpl.java:546)
at org.jboss.mx.util.MBeanProxyExt.invoke(MBeanProxyExt.java:177)
at $Proxy6.deploy(Unknown Source)
at org.jboss.deployment.scanner.URLDeploymentScanner.deploy(URLDeploymentScanner.java:302)
at org.jboss.deployment.scanner.URLDeploymentScanner.scan(URLDeploymentScanner.java:476)
at org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.doScan(Abstract
DeploymentScanner.java:201)
at org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.loop(Abstract
DeploymentScanner.java:212)
at org.jboss.deployment.scanner.AbstractDeploymentScanner$ScannerThread.run(Abstract
DeploymentScanner.java:191)
15:00:42,979 INFO [EjbMBeanAdaptor] Begin invoke, actionName=getState
15:00:45,778 INFO [Chap2EchoInfoBean] Created table 'CHAP2ECHOINFOBEAN' successfully.
15:00:45,924 INFO [EntityInstancePool] Started jboss.j2ee:jndiName=local/Chap2EchoInfoBean,plugin=
pool,service=EJB
15:00:45,929 INFO [EntityContainer] Started jboss.j2ee:jndiName=local/Chap2EchoInfoBean,service=EJB
15:00:46,025 INFO [StatelessSessionInstancePool] Started jboss.j2ee:jndiName=local/chap2.EchoBean,
plugin=pool,service=EJB
15:00:46,030 INFO [StatelessSessionContainer] Started jboss.j2ee:jndiName=local/chap2.EchoBean,
service=EJB
15:00:46,033 INFO [EjbMBeanAdaptor] Begin invoke, actionName=start
15:00:46,037 INFO [EjbMBeanAdaptor] Notified of start state
15:00:46,051 INFO [EjbMBeanAdaptor] Testing Echo
15:00:46,313 INFO [EchoBean] echo, info=echo info, arg=, arg=startService
15:00:46,319 INFO [EjbMBeanAdaptor] echo(startService) = startService
15:00:46,366 INFO [EjbMBeanAdaptor] Started null
15:00:46,370 INFO [EjbMBeanAdaptor] End invoke, actionName=start
15:00:46,372 INFO [EjbModule] Started jboss.j2ee:module=chap2-ex3.jar,service=EjbModule
15:00:46,375 INFO [EJBDeployer] Deployed: file:/private/tmp/jboss-3.2.3/server/default/tmp/deploy/
tmp58329chap2-ex3.ear-contents/chap2-ex3.jar
15:00:46,599 INFO [EARDeployer] Started J2EE application: file:/private/tmp/jboss-3.2.3/server/default/
```



```
deploy/chap2-ex3.ear
```

```
15:00:46,648 INFO [MainDeployer] Deployed package: file:/private/tmp/jboss-3.2.3/server/
default/deploy/chap2-ex3.ear
```

上述出现的栈调试信息（stack trace）并不是异常。位于 EjbMBeanAdaptor 第 150 行的代码触发了上述调试信息。当客户程序需要获得 MBean 提供的功能时，这种调试信息就能满足这种需求。请开发者注意，EJB 容器（标记为[EjbModule]行）先于实例 MBean（标记为[EjbMBeanAdaptor]行）启动。

接下来，请开发者使用 JMX 控制台 Web 应用来调用 echo 方法。浏览到页面：

```
http://localhost:8080/jmx-console/HtmlAdaptor?action=inspectMBean&name=jboss.book%
3Aservice%3DEjbMBeanAdaptor
```

并将页面滚动到 echo 操作入口，其视图如图 2-20 所示。

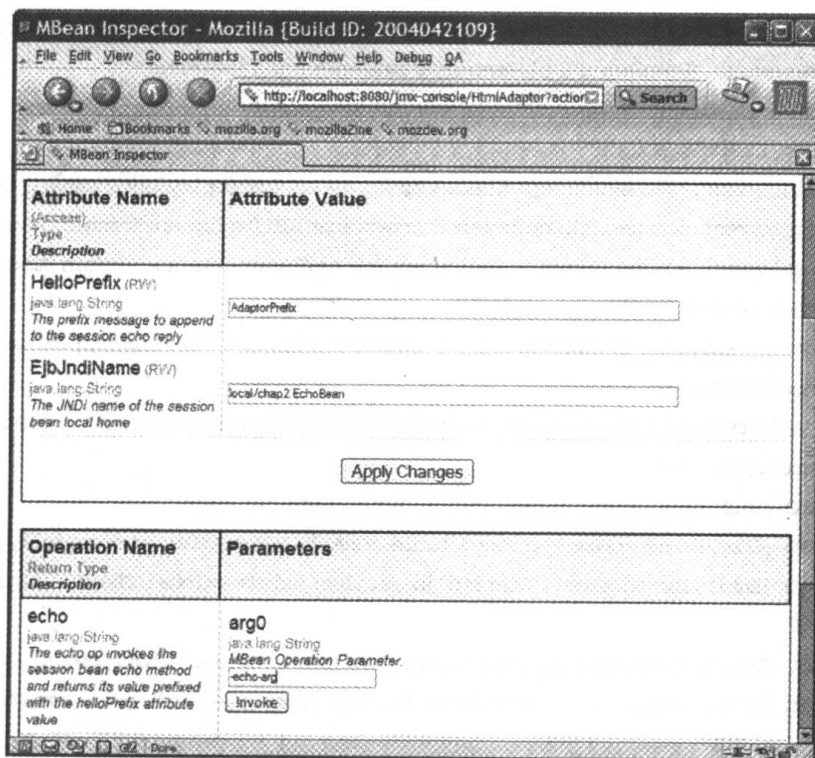


图 2-20 JMX 控制台视图中的 EjbMBeanAdaptor MBean 操作

图 2-20 中，已经在 ParamValue 文本域输入了“-echo-arg”参数。单击【Invoke】按钮后，响应页面将显示“AdaptorPrefix-echo-arg”字符串。服务器控制台输出若干行调试信息，它们来自 JMX 控制台中不同元数据的查询结果。MBean invoke 方法调试结果如下：

```
10:51:48,671 INFO [EjbMBeanAdaptor] Begin invoke, actionName=echo
10:51:48,671 INFO [EjbMBeanAdaptor] Lookup EchoLocalHome@local/chap2.EchoBean
10:51:48,687 INFO [EchoBean] echo, info=echo info, arg=, arg=-echo-arg
10:51:48,687 INFO [EjbMBeanAdaptor] Result: AdaptorPrefix-echo-arg
10:51:48,687 INFO [EjbMBeanAdaptor] End invoke, actionName=echo
```

2.5 JBoss 部署器架构

JBoss 具备可扩展的部署架构，使得开发者能够将不同的组件集成到 JBoss JMX 微内核中。图 2-21 展示了部署层的类结构。

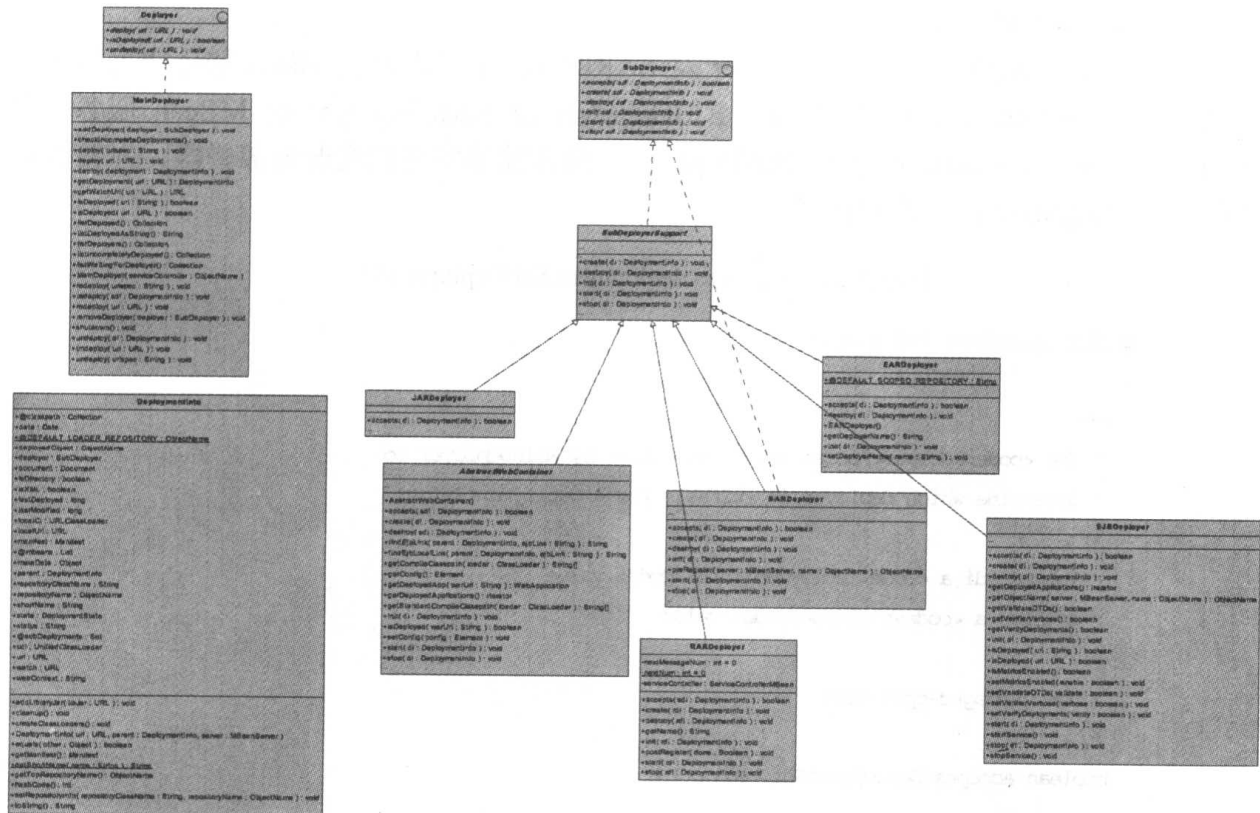


图 2-21 部署层类

MainDeployer 是部署入口点。组件部署请求发送给它，然后由它决定是否是否存在能够处理该部署单元的子部署器。如果存在，则将该部署工作委派于它。本文后续内容将演示 MainDeployer 如何使用 SARDeployer 部署 MBean 服务。JBoss 目前包含的部署器如下。

- **AbstractWebContainer**: 该子部署器处理 Web 应用存档(WAR)。AbstractWebContainer 处理后缀名为“war”的部署存档和目录。WAR 必须提供 WEB-INF/web.xml 描述符和可选的 WEB-INF/jboss-web.xml 描述符。
- **EARDeployer**: 该子部署器处理企业应用存档(EAR)。EARDeployer 处理后缀名为“ear”的部署存档和目录。EAR 必须提供 META-INF/application.xml 描述符和可选的 META-INF/jboss-app.xml 描述符。
- **EJBDeployer**: 该子部署器处理企业 Bean jar 包。它处理后缀名为“jar”的部署存档和目录。EJB jar 必须提供 META-INF/ejb-jar.xml 描述符和可选的 META-INF/jboss.xml 描述符。
- **JARDeployer**: 该子部署器处理库 jar 存档。其惟一的约束条件是，不能够含有

WEB-INF 目录。

- RARDeployer: 该子部署器处理 JCA 资源存档 (RAR)。RARDeployer 处理后缀名为 “rar” 的部署存档和目录。RAR 必须提供 META-INF/ra.xml 描述符。
- SARDeployer: 该子部署器处理 JBoss MBean 服务存档 (SAR)。它处理后缀名为 “sar” 的部署存档和目录。另外, SARDeployer 还处理以 “service.xml” 为后缀名的单独 XML 文件。对于存在 jar 库的 SAR 必须提供 META-INF/jboss-service.xml 描述符。

其中, MainDeployer、JARDeployer 及 SARDeployer 硬编码在 JBoss 服务器内核中。AbstractWebContainer、EARDeployer、EJBDeployer 及 RARDeployer 是 MBean 服务, 通过 MainDeployer 的 addDeployer (SubDeployer) 操作能够将它们注册为部署器。列表 2-34 给出了 SubDeployer 接口的源代码:

列表 2-34 org.jboss.deployment.SubDeployer 接口

```
public interface SubDeployer
{
    /**
     * The <code>accepts</code> method is called by MainDeployer to
     * determine which deployer is suitable for a DeploymentInfo.
     *
     * @param sdi a <code>DeploymentInfo</code> value
     * @return a <code>boolean</code> value
     *
     * @jmx:managed-operation
     */
    boolean accepts(DeploymentInfo sdi);

    /**
     * The <code>init</code> method lets the deployer set a few properties
     * of the DeploymentInfo, such as the watch url.
     *
     * @param sdi a <code>DeploymentInfo</code> value
     * @throws DeploymentException if an error occurs
     *
     * @jmx:managed-operation
     */
    void init(DeploymentInfo sdi) throws DeploymentException;

    /**
     * Set up the components of the deployment that do not
     * refer to other components
     *
     * @param sdi a <code>DeploymentInfo</code> value
     * @throws DeploymentException Failed to deploy
     */
}
```



```

    * @jmx:managed-operation
    */
    void create(DeploymentInfo sdi) throws DeploymentException;

    /**
     * The <code>start</code> method sets up relationships with other components.
     *
     * @param sdi a <code>DeploymentInfo</code> value
     * @throws DeploymentException if an error occurs
     *
     * @jmx:managed-operation
     */
    void start(DeploymentInfo sdi) throws DeploymentException;

    /**
     * The <code>stop</code> method removes relationships between components.
     *
     * @param sdi a <code>DeploymentInfo</code> value
     * @throws DeploymentException if an error occurs
     *
     * @jmx:managed-operation
     */
    void stop(DeploymentInfo sdi) throws DeploymentException;

    /**
     * The <code>destroy</code> method removes individual components
     *
     * @param sdi a <code>DeploymentInfo</code> value
     * @throws DeploymentException if an error occurs
     *
     * @jmx:managed-operation
     */
    void destroy(DeploymentInfo sdi) throws DeploymentException;
}

```

大体上, `DeploymentInfo` 对象具有如下数据结构, 即封装了可部署组件的完整状态。当 `MainDeployer` 收到部署请求时, 它迭代注册的子部署器, 并调用子部署器的 `accepts(DeploymentInfo)` 方法。 `MainDeployer` 将选中第一个 `accepts` 方法返回结果为 `true` 的子部署器, 并将 `init`、`create`、`start`、`stop` 及 `destroy` 部署生命周期操作委派给选中的子部署器。

部署器和类装载器

部署器实现了这样一种机制, 即将组件装载在 JBoss 服务器中。同时, 它也是大部分 UCL 实例的创建者, `MainDeployer` 是其中的主要创建者。在 `MainDeployer` 中 `init` 方法的

早期阶段, 它为部署组件。通过调用 `DeploymentInfo.createClassLoaders()` 的方法创建 UCL。从 JBoss 3.0.5RC1 开始, 实际上只有顶端的 `DeploymentInfo` 能够创建 UCL。所有的子部署操作将各自的类路径添加到它们的双亲 `DeploymentInfo` UCL 中。在 JBoss 3.0.5RC1 之前, 各个子部署操作都为部署任务创建 UCL, 而且为各个 manifest 或 classpath 引用创建单独的 UCL。这样很容易导致问题, 因为相同的类有可能被不同的 UCL 装载, 从而出现 `IllegalAccessError` 和 `LinkageError` 错误。各个部署组件都存在单独的 `URLClassLoader`, 并使用部署 URL 作为其路径, 从而可以定位到相应的资源, 比如部署描述符。图 2-22 给出了 `Deployer`、`DeploymentInfo` 及类装载器之间的交互关系。

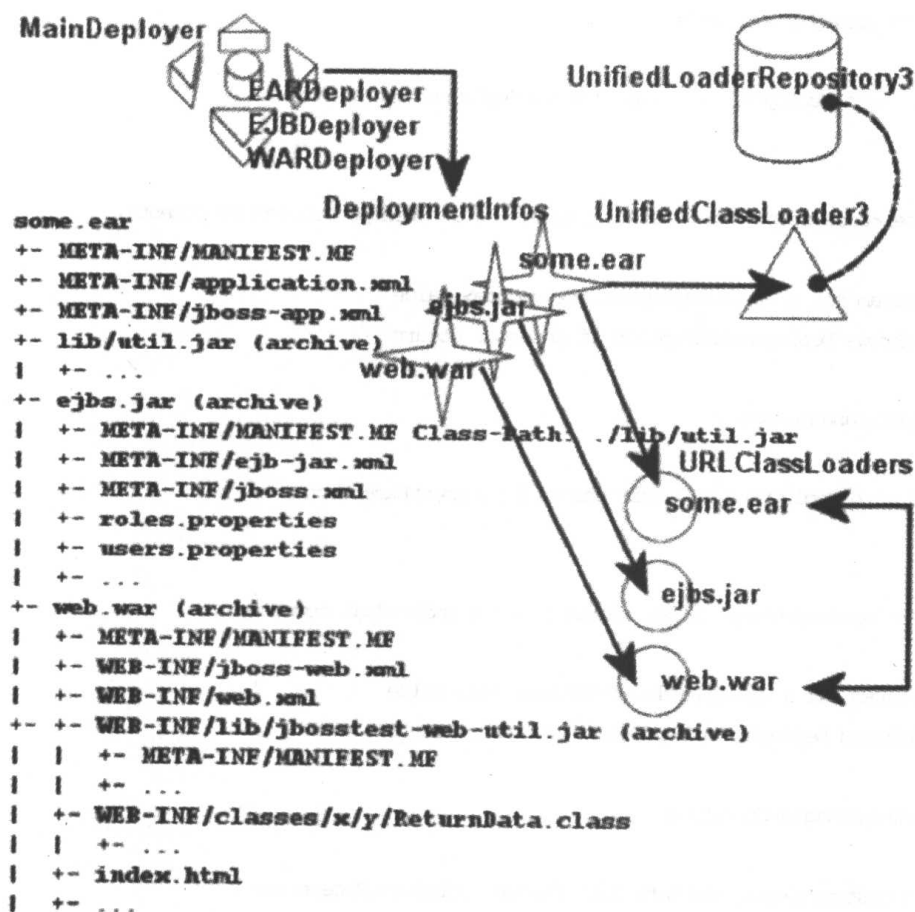


图 2-22 某 EAR 部署过程涉及到的类装载器

图 2-22 给出了包含 EJB 和 WAR 子部署单元的 EAR 部署组件。其中, EJB 部署单元借助于 manifest 引用了 lib/util.jar 实用 jar、WAR 在 WEB-INF/classes 目录下包含相应的类和 WEB-INF/lib 目录下的 jbosstest-web-util.jar 存档。各个部署都存在 `DeploymentInfo` 实例, 即这些实例都含有 `URLClassLoader` 指向各自的部署存档。其中, 和 `some.ear` 关联的 `DeploymentInfo` 是能够创建 UCL 的惟一候选者。ejb.jar 和 web.war 的 `DeploymentInfo` 将各自的部署存档添加给 `some.ear` 的 UCL classpath, 并共享该 UCL 作为它们的部署 UCL。EJBDeployer 将 manifest 中指定的任何 jar 添加给 EAR UCL。但对于 WARDeployer 而言, 情况有点特殊, 因为它仅仅将 WAR 存档添加给 `DeploymentInfo` UCL 的 classpath。Servlet 容器类装载器处理 WAR 中 WEB-INF/classes 和 WEB-INF/lib 中的类装载, 即将 WAR

DeploymentInfo UCL 作为其双亲类装载器。但是 Servlet 容器类装载器并不是 JBoss 类装载器库的一部分, 因此, WAR 内部的类对其他组件并不可见。如果需要在 Web 应用组件和其他组件, 比如 EJB 和 MBean, 之间共享类, 则必须将这些共享类装载到共享类装载器库中。具体的方式可以通过将类包含在 SAR 或 EJB 部署组件中, 或者通过 manifest 中的 Class-Path 入口指定引用包含共享类的 jar。对于 SAR 的情形, SAR 服务描述符中的 classpath 元素充当了 jar 中 manifest Class-Path 相同的作用。

2.6 借助于 SNMP 展示 MBean 事件

JBoss 3.2.2 发布版新增了 snmp-adaptor 服务, 它用于拦截 MBean 提交的 JMX 通知, 并将它们转换成 SNMP 陷阱 (Trap), 然后发送到 SNMP 管理器。据此, 开发者可以认为 snmp-adaptor 充当了 SNMP 代理的角色。snmp-adaptor 后续版本将支持完整的代理 get/set 功能, 并将其映射到 MBean 属性或操作上。

开发者可以将 JBoss 和更高级的系统 (网络) 管理平台 (如 HP OpenView) 集成, 从而将 MBean 功能展示给它们。MBean 开发者可以为任何重要事件 (如服务器冷启动) 生成通知, 以装备 MBean。配置的适配器能够拦截并映射通知给 SNMP 陷阱。该适配器将 OpenNMS 提供的 JoeSNMP 包作为其 SNMP 引擎。

SnpAgentService 是实现 SNMP 代理的主要 MBean。通过如下 3 个不同配置文件来配置该 MBean 服务:

- **managers.xml**: 配置发送 SNMP 陷阱的目的地。
- **mbeans.xml**: 配置监控的 MBean/通知类型。
- **notifications.xml**: 将各个通知类型指定到对应 SNMP 陷阱的精确映射。

2.6.1 SNMP 适配器服务

org.jboss.jmx.adaptor.snmp.agent.SnpAgentService 允许发送 V1 或 V2 SNMP 陷阱到一个或多个 SNMP 管理器。这些 SNMP 管理器由 IP 地址、监听端口及期望的 SNMP 版本定义。该服务包含的属性如下。

- **HeartBeatPeriod**: 用于生成 heartbeat 通知的周期 (单位: 秒)。
- **ManagersResName**: 用于指定文件资源名。该文件含有 SNMP 管理器规范。图 2-23 给出了该文件的内容模型。
- **MonitoredObjectsResName**: 用于指定文件资源名。该文件配置了用于监控事件的 JMX 对象。图 2-24 给出了该文件的内容模型。
- **NotificationMapResName**: 指定文件资源名。该文件含有 JMX 通知到 SNMP 陷阱的映射。图 2-25 给出了该文件的内容模型。
- **TrapFactoryClassName**: 用于指定 org.jboss.jmx.adaptor.snmp.agent.TrapFactory 实现类。该实现类负责将 JMX 通知转换成 V1 和 V2 SNMP 陷阱。
- **TimerName**: 指定 JMX 定时服务的 JMX ObjectName。该 JMX 定时服务供 heartbeat

通知使用。



图 2-23 SNMP managers.xml 文件的内容模型

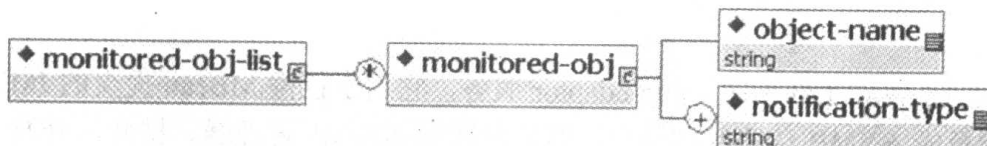


图 2-24 内容模型（用于监控对象文件）

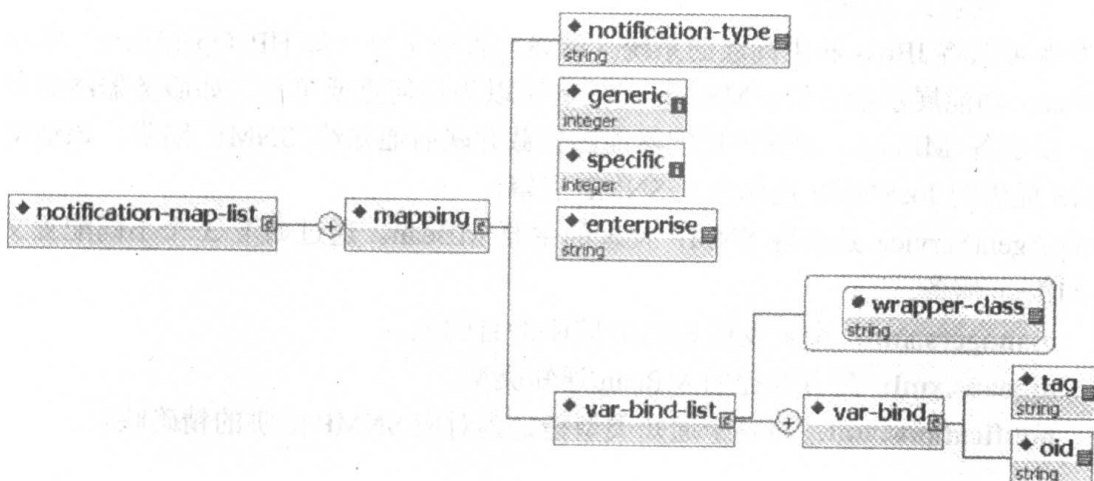


图 2-25 内容模型（完成通知到陷阱映射的任务）

2.6.2 陷阱服务事件

org.jboss.jmx.adaptor.snmp.trapd.TrapdService 是担任 SNMP 管理器的简单 MBean。该 MBean 服务监听可配置端口是否有陷阱到来，并使用系统日志器将这些陷阱以 DEBUG 消息的形式记录下来。开发者可以修改 Log4j 配置，将日志输出重定向到文件。SnmpAgentService 和 TrapdService 服务之间没有相互依赖关系。

2.7 远程访问服务——分离式 Invoker

除了能够为 MBean 服务集成所需功能外，JBoss 还提供了分离式 Invoker 的概念，即允许 MBean 服务借助于特定协议将功能接口展示给远程客户，从而供它们访问。JBoss 3.0 首次为 EJB 容器引入这种概念，从 JBoss 3.2 开始 Invoker 概念已经被广泛应用于任何 MBean

服务。分离式 Invoker 的含义在于，这里提及的远程和服务访问协议只是从功能层面考虑的，或者说它是独立于 MBean 服务组件的。因此，开发者可以借助于 RMI/JRMP、RMI/HTTP、RMI/SOAP 或任何定制协议将其命名服务展示给远程客户。

接下来首先讨论分离式 Invoker 架构，将给出涉及组件的初步介绍。图 2-26 给出了分离式 Invoker 架构中的主要组件。

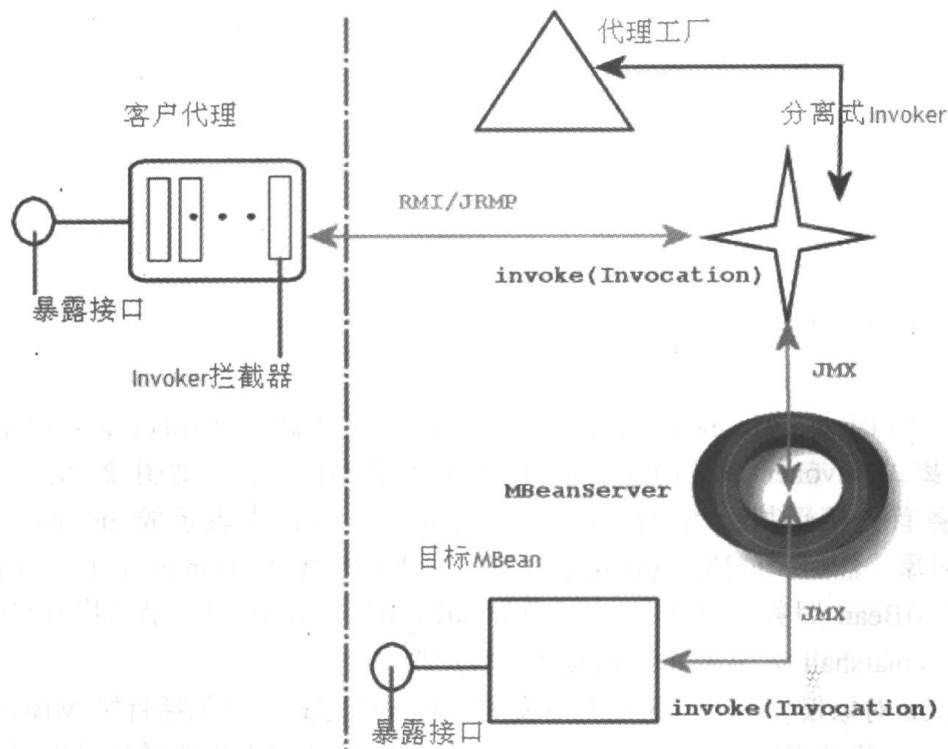


图 2-26 分离式 Invoker 架构的主要组件

从图 2-26 可以看出，在客户端存在客户代理，它暴露了 MBean 服务的接口。这也是 JBoss 用于 EJB home 和远程接口的精炼、较少编译动态代理。但对于这两种场合，其惟一的不同之处在于展示的接口集合及代理中客户端拦截器的不同。客户端拦截器在图中是使用矩形表示的（客户代理部分中）。拦截器是一种用于转换方法调用和（或）返回值的流水线类型模式。客户使用查找机制获得代理，通常情况下都是借助于 JNDI 获得的。尽管图 2-26 标明的是使用 RMI 方式，但实际上只要求展示的接口及其类型在通过 JNDI 和传输层时满足可序列化条件。

传输层的选择完全取决于客户代理中最后一个拦截器，即图 2-26 中标明“Invoker 拦截器”的矩形。其中，Invoker 拦截器含有对服务器端“分离式 Invoker”MBean 服务的具体传输存根的引用。Invoker 拦截器也处理位于目标 MBean 中同一 JVM 的调用优化工作。当 Invoker 拦截器发现调用是通过传址的方式时，它将简单地将该调用传递给目标 MBean。

分离式 Invoker 服务要负责借助于传输层生成通用的调用操作。列表 2-35 给出了 org.jboss.invocation.Invoker 接口，即通用的调用操作语法规则。

列表 2-35 Invoker 接口

```
package org.jboss.invocation;

import java.rmi.Remote;
import org.jboss.proxy.Interceptor;
import org.jboss.util.id.GUID;

public interface Invoker
    extends Remote
{
    GUID ID = new GUID();

    Object invoke(Invocation invocation) throws Exception;
}
```

Invoker 接口继承于 Remote，以兼容于 RMI，但并不意味 Invoker 必须暴露其 RMI 服务存根。分离式 Invoker 服务简单地充当了传输网关的作用。正如图 2-26 所示，分离式 Invoker 服务首先接受基于特定传输层的调用请求，该请求表示成 org.jboss.invocation.Invocation 对象。然后，分离式 Invoker 服务完成调用的解包（unmarshall），并将调用请求发送给目的 MBean 服务（图 2-26 中“目标 MBean”部分所示），最后将返回值或异常信息进行压包（marshall），从而将结果发送给客户端。

Invocation 对象仅仅是方法调用上下文的一种表达方式。它包括目标 MBean 名字、方法、方法参数、代理工厂为代理附加的上下文信息，以及客户代理拦截器为调用附加的其他数据信息。列表 2-36 给出了 Invocation 类的主要方法。

列表 2-36 用于表达方法调用请求的 Invocation 类

```
package org.jboss.invocation;

import java.lang.reflect.Method;
import java.security.Principal;
import java.util.Map;
import java.util.HashMap;
import javax.transaction.Transaction;

public class Invocation
{
    /** The signature of the invoke() method */
    public static final String[] INVOKE_SIGNATURE = {"org.jboss.invocation.Invocation"};

    /**
     * Contextual information to the invocation that is not part of the payload.
     */
    public Map transient_payload;
```

```
/**
 * as_is classes that will not be marshalled by the invocation
 * (java.* and javax.* or anything in system classpath is OK)
 */
public Map as_is_payload;

/** Payload will be marshalled for type hiding at the RMI layers. */
public Map payload;
protected InvocationContext invocationContext;
protected Object[] args;
protected Object objectName;
protected Method method;

public Invocation()
{
    payload = new HashMap();
    as_is_payload = new HashMap();
    transient_payload = new HashMap();
}

public Invocation( Object id, Method m, Object[] args, Transaction tx,
    Principal identity, Object credential )
{
    this.payload = new HashMap();
    this.as_is_payload = new HashMap();
    this.transient_payload = new HashMap();

    setId(id);
    setMethod(m);
    setArguments(args);
    setTransaction(tx);
    setPrincipal(identity);
    setCredential(credential);
}

public void setValue(Object key, Object value)
{
    setValue(key, value, PayloadKey.PAYLOAD);
}

public void setValue(Object key, Object value, PayloadKey type)
{
    if(type == PayloadKey.TRANSIENT)
    {
        transient_payload.put(key,value);
    }
}
```



```
        else if(type == PayloadKey.AS_IS)
        {
            as_is_payload.put(key,value);
        }
        else if(type == PayloadKey.PAYLOAD)
        {
            payload.put(key,value);
        }
        else
        {
            throw new IllegalArgumentException("Unknown PayloadKey: " + type);
        }
    }

    public Object getValue(Object key)
    {
        // find where it is
        Object rtn = payload.get(key);
        if (rtn != null) return rtn;

        rtn = as_is_payload.get(key);
        if (rtn != null) return rtn;

        rtn = transient_payload.get(key);
        return rtn;
    }

    public Object getPayloadValue(Object key)
    {
        return payload.get(key);
    }

    ... Convenience accessor methods deleted...
}
```

图 2-26 中“代理工厂”组件表明，客户代理的配置是由服务器端代理工厂 MBean 服务完成的。其中，代理工厂完成如下任务：

- 创建动态代理，其实现了目标 MBean 打算暴露的接口。
- 使客户代理拦截器与动态代理处理器产生关联。
- 将动态代理和调用请求上下文产生关联。该过程包括目标 MBean、分离式 Invoker 存根及代理 JNDI 名字。
- 通过将代理绑定于 JNDI 上，使得客户端能够访问到它。

本文待介绍的最后一个组件为图 2-26 中的“目标 MBean”服务，它暴露了供远程客户访问的所需接口。为使通过指定接口访问 MBean 服务成为可能，开发者需要完成如下几步。

步骤

(1) 定义 JMX 操作对应方法的语法规则:

```
public Object invoke(org.jboss.invocation.Invocation) throws Exception
```

(2) 使用 `org.jboss.invocation.MarshalledInvocation.calculateHash` 方法创建 `HashMap` `<Long, Method>` 映射, 即通过 `java.lang.reflect.Method` 获得的暴露接口映射成 `Long` 型的 `Hash` 表示。

(3) 实现 `invoke (Invocation)` JMX 操作, 并使用上述 `Hash` 表示转换成暴露接口的 `java.lang.reflect.Method`。其中, 反射完成了对对象的实际调用操作, 而该对象关联到实际实现了暴露接口的 `MBean` 服务。

2.7.1 分离式 Invoker 实例: MBeanServer Invoker 适配器服务

本文在“连接到 JMX 服务器”节内容, 已经提到能够使用 `Invoker` 服务, 并借助于任何协议访问 `javax.management.MBeanServer`。本小节将阐述这样一种 `Invoker` 服务, 即 `org.jboss.jmx.connector.invoker.InvokerAdaptorService`。同时, 本文为演示如何远程访问 `MBean` 服务, 还给出了借助于 `RMI/JRMP` 访问的配置信息。

`InvokerAdaptorService` 是简单 `MBean` 服务, 它只是完成分离式 `Invoker` 模式中的目标 `MBean` 角色。见列表 2-37。

列表 2-37 InvokerAdaptorService MBean

```
615 package org.jboss.jmx.connector.invoker;
616 public interface InvokerAdaptorServiceMBean
617     extends org.jboss.system.ServiceMBean
618 {
619     Class getExportedInterface();
620     void setExportedInterface(Class exportedInterface);
621
622     Object invoke(org.jboss.invocation.Invocation invocation)
623         throws Exception;
624 }

625 package org.jboss.jmx.connector.invoker;
626
627 import java.lang.reflect.InvocationTargetException;
628 import java.lang.reflect.Method;
629 import java.lang.reflect.UndeclaredThrowableException;
630 import java.util.Collections;
631 import java.util.HashMap;
632 import java.util.Map;
633
634 import javax.management.MBeanServer;
```

```
635 import javax.management.ObjectName;
636
637 import org.jboss.invocation.Invocation;
638 import org.jboss.invocation.MarshalledInvocation;
639 import org.jboss.mx.server.ServerConstants;
640 import org.jboss.system.ServiceMBeanSupport;
641 import org.jboss.system.Registry;
642
643 public class InvokerAdaptorService
644     extends ServiceMBeanSupport
645     implements InvokerAdaptorServiceMBean, ServerConstants
646 {
647     private static ObjectName mbeanRegistry;
648
649     static
650     {
651         try
652         {
653             mbeanRegistry = new ObjectName(MBEAN_REGISTRY);
654         }
655         catch (Exception e)
656         {
657             throw new RuntimeException(e.toString());
658         }
659     }
660
661     private Map marshalledInvocationMapping = new HashMap();
662     private Class exportedInterface;
663
664     public Class getExportedInterface()
665     {
666         return exportedInterface;
667     }
668
669     public void setExportedInterface(Class exportedInterface)
670     {
671         this.exportedInterface = exportedInterface;
672     }
673
674     protected void startService()
675         throws Exception
676     {
677         // Build the interface method map
678         Method[] methods = exportedInterface.getMethods();
679         HashMap tmpMap = new HashMap(methods.length);
680         for(int m = 0; m < methods.length; m ++)
```

```
681 {
682     Method method = methods[m];
683     Long hash = new Long(MarshalledInvocation.calculateHash(method));
684     tmpMap.put(hash, method);
685 }
686 marshalledInvocationMapping = Collections.unmodifiableMap(tmpMap);
687 // Place our ObjectName hash into the Registry so invokers can resolve it
688 Registry.bind(new Integer(serviceName.hashCode()), serviceName);
689 }
690
691 protected void stopService()
692     throws Exception
693 {
694     Registry.unbind(new Integer(serviceName.hashCode()));
695 }
696
697
698 public Object invoke(Invocation invocation)
699     throws Exception
700 {
701     // Make sure we have the correct classloader before unmarshalling
702     Thread thread = Thread.currentThread();
703     ClassLoader oldCL = thread.getContextClassLoader();
704
705     // Get the MBean this operation applies to
706     ClassLoader newCL = null;
707     ObjectName objectName = (ObjectName) invocation.getValue("JMX_OBJECT_NAME");
708     if (objectName != null)
709     {
710         // Obtain the ClassLoader associated with the MBean deployment
711         newCL = (ClassLoader) server.invoke
712         (
713             mbeanRegistry, "getValue",
714             new Object[] { objectName, CLASSLOADER },
715             new String[] { ObjectName.class.getName(), "java.lang.String" }
716         );
717     }
718
719     if (newCL != null && newCL != oldCL)
720         thread.setContextClassLoader(newCL);
721
722     try
723     {
724         // Set the method hash to Method mapping
725         if (invocation instanceof MarshalledInvocation)
726         {
```

```
727     MarshalledInvocation mi = (MarshalledInvocation) invocation;
728     mi.setMethodMap(marshalledInvocationMapping);
729 }
730 // Invoke the MBeanServer method via reflection
731 Method method = invocation.getMethod();
732 Object[] args = invocation.getArguments();
733 Object value = null;
734 try
735 {
736     String name = method.getName();
737     Class[] sig = method.getParameterTypes();
738     Method mbeanServerMethod = MBeanServer.class.getMethod(name, sig);
739     value = mbeanServerMethod.invoke(server, args);
740 }
741 catch(InvocationTargetException e)
742 {
743     Throwable t = e.getTargetException();
744     if( t instanceof Exception )
745         throw (Exception) t;
746     else
747         throw new UndeclaredThrowableException(t, method.toString());
748 }
749
750 return value;
751 }
752 finally
753 {
754     if (newCL != null && newCL != oldCL)
755         thread.setContextClassLoader(oldCL);
756 }
757 }
758 }
```

接下来将仔细分析上述列表中的重点内容。InvokerAdaptorService 的标准 MBean 接口 InvokerAdaptorServiceMBean 由单个 ExportedInterface 属性和单个 invoke(Invocation)操作构成。ExportedInterface 属性允许自定义暴露给客户的接口类型。同时，它的方法名和语法规则必须“兼容”于 MBeanServer 类。为参与到分离式 Invoker 模式中，开发者必须实现目标 MBean 服务的入口，即 invoke (Invocation) 操作。该操作由分离式 Invoker 服务调用，其中也需要将 Invoker 服务配置成有权访问 InvokerAdaptorService。

第 54~64 行，org.jboss.invocation.MarshalledInvocation.calculateHash(Method)实用方法创建了 ExportedInterface 类的 HashMap<Long, Method>。但由于实例 java.lang.reflect.Method 不是序列化的，因此使用了未序列化 Invocation 类的 MarshalledInvocation 版本实现完成客户端和服务端之间调用的压包工作。MarshalledInvocation 将 Method 实例替换成对应的 Hash 表示。在服务器端，必须告知 MarshalledInvocation 提供给 Method 映射的 Hash 值。

第 64 行创建了 `InvokerAdaptorService` 服务名和其 Hash 码表示之间的映射。这使得分离式 `Invoker` 能够知道 `Invocation` 所要调用目标 `MBean` 的 `ObjectName`。当目标 `MBean` 名存储在 `Invocation` 中时, 由于创建 `ObjectName` 对象很耗系统资源, 因此通常都存储为 Hash 码。`org.jboss.system.Registry` 完成全局映射, 比如 `Invoker` 使用它存储 `ObjectName` 映射后的 Hash 码。

第 77~93 行获得 `MBeanServer` 上待操作的 `MBean` 名, 并查找部署该 `MBean` SAR 的相关类装载器。借助 `org.jboss.mx.server.registry.BasicMBeanRegistry` 能够获得所需信息, 它是 JBoss JMX 具体实现类。由于分离式 `Invoker` 协议层并不能访问寻找到的类加载器, 因此为完成调用请求相关类型的解包工作, 通常情况下都需要为 `MBean` 创建正确的类装载上下文。

第 101~105 行, 如果调用请求的参数是 `MarshaledInvocation` 类型, 则这些行将 `ExposedInterface` 类的方法 Hash 设置成方法映射。这里的方法映射计算规则使用了前面第 54~62 行所使用的。

第 107~114 行完成 `ExposedInterface` Method 到 `MBeanServer` 类对应方法的再次映射。`InvokerAdaptorService` 通过引入额外接口而降低了 `ExposedInterface` 与 `MBeanServer` 的耦合度。由于标准 `java.lang.reflect.Proxy` 类只能代理接口, 所以从该角度考虑, 该额外接口是必需的。另外, 这也使得仅暴露 `MBeanServer` 的部分方法成为可能, 它还允许往 `ExposedInterface` 方法定义添加具体传输异常信息, 比如 `java.rmi.RemoteException`。

第 115 行, 分发 `MBeanServer` 方法调用请求给部署了 `InvokerAdaptorService` 的 `MBeanServer` 实例。其中, 服务器实例变量继承于其父类 `ServiceMBeanSupport`。

第 117~124 行处理来自反射调用操作抛出的任何异常。这些行也处理那些由调用操作抛出的未包裹 (unwrapping)、任何已声明异常。

第 126 行, 成功返回 `MBeanServer` 方法调用结果。

请注意, `InvokerAdaptorService` `MBean` 并没有直接涉及到任何的具体传输细节。尽管存在方法 Hash 到 Method 映射的计算, 但这独立于传输细节。

至此, 本文开始介绍如何使用 `InvokerAdaptorService` `MBean` 服务, 并借助于 RMI/JRMP 暴露, “2.3.2 使用 RMI 连接到 JMX” 一节提到的, `org.jboss.jmx.adaptor.rmi.RMIAdaptor` 接口。其中, `default` 配置文件集合中的 `jmx-invoker-adaptor-service.sar` 部署组件给出了代理工厂和 `InvokerAdaptorService` 配置。列表 2-38 给出了用于该部署组件的 `jboss-service.xml` 描述符。

列表 2-38 default jmx-invoker-adaptor-server.sar 中的 jboss-service.xml 描述符

```
759 <server>
760 <!-- The JRMP invoker proxy configuration for the InvokerAdaptorService -->
761 <mbean code="org.jboss.invocation.jmp.server.JRMPProxyFactory"
762 name="jboss:jmx:type=adaptor,name=Invoker,protocol=jmp,service=proxyFactory">
763 <!-- Use the standard JRMPInvoker from conf/jboss-service.xml -->
764 <attribute name="InvokerName">jboss:service=invoker,type=jrmp</attribute>
765 <!-- The target MBean is the InvokerAdaptorService configured below -->
766 <attribute name="TargetName">jboss:jmx:type=adaptor,name=Invoker</attribute>
```

```
767 <!-- Where to bind the RMIAdaptor proxy -->
768 <attribute name="JndiName">jmx/invoker/RMIAdaptor</attribute>
769 <!-- The RMI compabitle MBeanServer interface -->
770 <attribute name="ExportedInterface">org.jboss.jmx.adaptor.rmi.RMIAdaptor
771 </attribute>
772 <attribute name="ClientInterceptors">
773   <iterceptors>
774     <interceptor>org.jboss.proxy.ClientMethodInterceptor</interceptor>
775   </interceptor>
776   org.jboss.jmx.connector.invoker.client.InvokerAdaptorClientInterceptor
777 </interceptor>
778   <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
779 </iterceptors>
780 </attribute>
781 <depends>jboss:service=invoker,type=jrmp</depends>
782 </mbean>
783
784 <!-- This is the service that handles the RMIAdaptor invocations by routing
785 them to the MBeanServer the service is deployed under. -->
786 <mbean code="org.jboss.jmx.connector.invoker.InvokerAdaptorService"
787   name="jboss.jmx:type=adaptor,name=Invoker">
788   <attribute name="ExportedInterface">org.jboss.jmx.adaptor.rmi.RMIAdaptor
789   </attribute>
790 </mbean>
791 </server>
```

第一个 MBean, org.jboss.invocation.jrmp.server.JRMPProxyFactory, 是能够为 RMI/JRMP 协议创建代理的代理工厂 MBean 服务。在 2.7.2 中的“4. JRMPProxyFactory 服务——创建动态 JRMP 代理”有 JRMPProxyFactory 的完整介绍。开发者从上述列表可以看出, JRMPInvoker 为分离式 Invoker, InvokerAdaptorService 为接受调用请求的目标 MBean。其中, 代理将暴露 RMIAdaptor 接口, 并且该接口将被绑定在 JNDI, 即“jmx/invoker/RMIAdaptor”上。另外, 代理含有 3 个拦截器: ClientMethodInterceptor、InvokerAdaptorClientInterceptor 及 InvokerInterceptor。InvokerAdaptorService 服务简单地通过 RMIAdaptor 接口方式暴露出来。

配置的最后一部分给出了分离式 Invoker 的配置, 即借助于 RMI/JRMP 暴露 Invoker AdaptorService。这里使用的分离式 Invoker 是 EJB 容器供 home 和远程调用使用的标准 RMI/JRMP Invoker, 即 org.jboss.invocation.jrmp.server.JRMPInvoker MBean 服务, 它配置于 conf/jboss-service.xml 描述符中。因此, 这里使用相同的服务实例更能体现 Invoker 的分离式特性。无论代理暴露了何种接口, 或代理使用了何种服务, JRMPInvoker 简单地地为所有 RMI/JRMP 代理充当了 RMI/JRMP 端点 (endpoint) 的作用。

2.7.2 分离式 Invoker 参考

1. JRMPInvoker - RMI/JRMP 传输

MBean 服务, 即 `org.jboss.invocation.jrmp.server.JRMPInvoker` 类提供了 Invoker 接口的 RMI/JRMP 实现。JRMPInvoker 将自身导出成 RMI 服务器, 因此当它作为远程客户的 Invoker 时, 需要将 JRMPInvoker 存根发送给客户, 并使用 RMI/JRMP 协议完成调用过程。

JRMPInvoker MBean 提供很多属性以配置 RMI/JRMP 传输层。具体属性如下。

- **RMIObjectPort:** 设置 RMI 服务器 Socket 监听端口。当通过代理接口进行通信时, RMI 客户端连接到该端口。jboss-service.xml 描述符给出的默认设置为 4444。如果未指定, 则属性默认值为 0, 即使用匿名端口。
- **RMIClientSocketFactory:** 指定 `java.rmi.server.RMIClientSocketFactory` 接口的全限定类名, 供导出代理接口期间使用。
- **RMI ServerSocketFactory:** 指定 `java.rmi.server.RMIServerSocketFactory` 接口的全限定类名, 供导出代理接口期间使用。
- **ServerAddress:** 指定 RMI 服务器 Socket 监听端口使用的接口地址。该地址可能为 DNS 主机名或 IP 地址的形式。由于 `RMIServerSocketFactory` 没有提供接收 `InetAddress` 对象的方法, 所以 `ServerAddress` 值是使用反射传递给 `RMIServerSocketFactory` 实现类的。检查方法 `public void setBindAddress (java.net.InetAddress addr)` 是否存在。如果存在, 则将 `RMIServerSocketAddr` 值传递给 `RMIServerSocketFactory` 实现类。但如果其实现类没有提供这样的方法, 则将忽略 `ServerAddress` 值。
- **SecurityDomain:** 指定 `org.jboss.security.SecurityDomain` 接口实现的 JNDI 名。该实现与 `RMIServerSocketFactory` 接口实现关联。通过反射将其取值传递给 `RMIServerSocketFactory` 以定位方法命名和语法规则为: `public void setSecurityDomain (org.jboss.security.SecurityDomain d)` 的方法。如果寻找不到, 则忽略现有的 `SecurityDomain` 取值。

2. PooledInvoker - RMI/套接字传输

MBean 服务, 即 `org.jboss.invocation.pooled.server.PooledInvoker` 类为 Invoker 接口提供了基于自定义套接字传输的 RMI 实现。PooledInvoker 将自身导出为 RMI 服务器, 因此当它作为远程客户的 Invoker 时, 需要将 PooledInvoker 存根发送给客户端, 并使用自定义套接字协议完成调用过程。

PooledMBean MBean 提供了很多属性以完成 Socket 传输层的配置。具体属性如下。

- **NumAcceptThreads:** 用于接收客户连接的线程数量。默认值为 1。
- **MaxPoolSize:** 用于处理客户的服务器线程数量。默认值为 300。
- **SocketTimeout:** 指定传递给 `Socket.setSoTimeout()` 方法的套接字超时值。默认值为 60 000。
- **ServerBindPort:** 指定服务器套接字的端口。如果使用 0 值, 则表示使用某匿名

端口。

- **ClientConnectAddress**: 指定客户传递给 `Socket(addr, port)` 构建器的地址。在服务器上运行如下方法, 默认为 `InetAddress.getLocalHost()` 的返回值。
- **ClientConnectPort**: 指定客户传递给 `Socket(addr, port)` 构建器的端口。默认值为服务器监听套接字的端口。
- **ClientMaxPoolSize**: 指定客户端线程的最大数量。默认值为 300。
- **Backlog**: 指定与服务器接收套接字关联的 backlog。默认值为 200。
- **EnableTcpNoDelay**: 为标志位, 它表明客户端套接字的 `TcpNoDelay` 标志是否有效。默认值为 `false`。
- **ServerBindAddress**: 指定服务器绑定的监听 `Socket`。默认为空值, 即服务器应该绑定在所有接口上。
- **TransactionManagerService**: 指定 JTA 事务管理器服务的 JMX ObjectName。

3. IIOPInvoker - RMI/IIOP 传输

MBean 服务, 即 `org.jboss.invocation.iiop.IIOPInvoker` 类提供了 `Invoker` 接口的 RMI/IIOP 实现。IIOPInvoker 将 IIOP 调用请求转发给 CORBA Servant, 即通过 `org.jboss.proxy.ejb.IORFactory` 代理工厂创建 RMI/IIOP 代理。然而, 该工厂并不是创建标准的 Java 代理 (正如 JRMP 代理工厂一样), 它创建 CORBA IOR。其中, `IORFactory` 将关联到某指定企业 Bean。它使用两个 CORBA Servant 能够注册 IIOP Invoker, 即用于企业 Bean `EJBHome` 的 `EjbHomeCorbaServant` 和用于企业 Bean `EJBObject` 的 `EjbObjectCorbaServant`。

IIOPInvoker MBean 没有提供可配置属性, 因为所有属性都是通过 JacORB CORBA 服务的 `conf/jacorb.properties` 属性文件配置完成的。

4. JRMPProxyFactory 服务——创建动态 JRMP 代理

MBean 服务, 即 `org.jboss.invocation.jrmp.server.JRMPProxyFactory` 类是代理工厂。它能够以兼容于 RMI 语义的形式暴露任何接口, 从而允许远程客户使用 JRMP 协议方式访问 MBean 服务。

JRMPProxyFactory 支持如下属性。

- **InvokerName**: 指定用于处理 RMI/JRMP 传输的服务器端 JRMPInvoker MBean 服务 JMX ObjectName 字符串。
- **TargetName**: 指定服务器端 MBean, 它为 `ExportedInterface` 暴露 `invoke` (`Invocation`) JMX 操作。任何通过该代理的访问操作都是使用 `TargetName` 值作为其目的 MBean 服务的。
- **JndiName**: 指定代理绑定到的 JNDI 名。
- **ExportedInterface**: 指定代理实现的接口的全限定类名。客户使用代理的类型化视图完成调用过程。
- **ClientInterceptors**: 指定 `interceptors/interceptor` 元素片段。各个 `interceptor` 元素内容指定了代理拦截器栈中 `org.jboss.proxy.Interceptor` 实现的全限定类名。`interceptors/interceptor` 元素的顺序限定了拦截器的顺序。

5. HttpInvoker - RMI/HTTP 传输

MBean 服务, 即 `org.jboss.invocation.http.server.HttpInvoker` 类为基于 HTTP 方式对 JMX 总线的调用提供支持。不同于 `JRMPInvoker`, `HttpInvoker` 不是 `Invoker` 实现, 但它实现了 `Invoker.invoke` 方法。JBoss 应用服务器在完成 HTTP POST 请求时, 通过 `org.jboss.invocation.http.servlet.InvokerServlet` 能够间接地访问到 `HttpInvoker`。`HttpInvoker` 以 `org.jboss.invocation.http.interfaces.HttpInvokerProxy` 类形式导出了客户端代理。该类是 `Invoker` 的实现, 并且是可序列化的。为替换 `JRMPInvoker`, 开发者可以将 `HttpInvoker` 作为 `bean-invoker` 和 `home-invoker` EJB 配置元素的目标值。`HttpInvoker` 和 `InvokerServlet` 部署在 `http-invoker.sar` 中。

“3.2.2 基于 HTTP 访问 JNDI”节内容有其更进一步讨论。

`HttpInvoker` 支持如下属性。

- **InvokerURL**: `InvokerURL` 属性是 `InvokerServlet` 映射的 HTTP URL 或系统属性名。客户端 JVM 将解析该系统属性名, 并获得 `InvokerServlet` 的 HTTP URL。如果 `InvokerURL` 取值是以 `{x}` 形式出现的, 其中 `x` 为系统属性名, 则该值本身能够引用服务器解析的系统属性, 因此能够在单个位置设置 URL 或客户端系统属性, 并能够在 `HttpInvoker` 和 `InvokerServlet` 配置中多次使用。
- **InvokerURLPrefix**: 如果没有设置 `InvokerURL`, 则将通过 `InvokerURLPrefix` + `localhost` + `InvokerURLSuffix` 组合构建它。比如, 默认的 `InvokerURLPrefix` 取值为 “`http://`”。
- **InvokerURLSuffix**: 如果没有设置 `InvokerURL`, 则将通过 `InvokerURLPrefix` + `localhost` + `InvokerURLSuffix` 组合构建它。比如, 默认的 `InvokerURLSuffix` 取值为 “`:8080/invoker/JMXInvokerServlet`”。
- **UseHostName**: `boolean` 标志位, 即应该使用 `InetAddress.getHostName()` 还是 `getHostAddress()` 方法的返回值作为上述 `localhost` 的值。如果为 `true`, 则使用 `getHostName()`, 否则使用 `getHostAddress()` 方法。

6. HA JRMPInvoker——群集 RMI/JRMP 传输

`org.jboss.proxy.generic.ProxyFactoryHA` 服务扩展了 `ProxyFactory`, 即它是群集敏感工厂。`ProxyFactoryHA` 完整地支持 `JRMPProxyFactory` 的所有属性。其含义为自定义端口、接口及 `Socket` 传输绑定对于群集 RMI/JRMP 可用。另外, `ProxyFactoryHA` 服务还支持如下具体群集属性。

- **PartitionObjectName**: 指定代理关联的群集服务的 JMX `ObjectName`。
- **LoadBalancePolicy**: `org.jboss.ha.framework.interfaces.LoadBalancePolicy` 接口实现的类名, 它与代理关联。

7. HA HttpInvoker——群集 RMI/HTTP 传输

在 JBoss-3.0.3 中, 已经扩展了 JBoss-3.0.2 添加的 RMI/HTTP 层, 使得其能够处理群集环境中软件调用的负载均衡。同时, 它在 `HttpInvoker` 基础上扩展了 HA 功能, 并从 HA-RMI/JRMP 群集借用了大量的功能。

为使用 HA-RMI/HTTP, 开发者需要为 EJB 容器配置 `Invoker`。通过 `jboss.xml` 描述符或 `standardjboss.xml` 描述符, 开发者能够实现该项功能的配置。在第 5 章的列表 5-7 中给

出了从 org.jboss.test.hello 测试套件包中取出的某无状态会话配置实例。

8. HttpProxyFactory——创建动态 HTTP 代理

MBean 服务，即 org.jboss.invocation.http.server.HttpProxyFactory 类是代理工厂。它使用 HTTP 传输方式的远程访问客户暴露兼容 RMI 语义的任何接口。

HttpProxyFactory 支持如下属性。

- **InvokerName**: 指定服务器端为 ExportedInterface 暴露 invoke 操作的 MBean。该名字（译者注：InvokerName 取值）嵌入在 HttpInvokerProxy 上下文中，并成为 HttpInvoker 前向调用的目标。
- **JndiName**: 指定 HttpProxyFactory 绑定到的 JNDI 名。客户端查找该名字，以获得动态代理。动态代理暴露了服务接口，并基于 HTTP 方式完成调用的压包操作。如果为空值，则表明不该将 HttpProxyFactory 绑定到 JNDI。
- **InvokerURL**: 指定 InvokerServlet 映射的 HTTP URL 或系统属性名。客户 JVM 将解析该系统属性名，并获得 InvokerServlet 的 HTTP URL。如果 InvokerURL 取值是以 \${x} 形式出现的，其中 x 为系统属性名，则该值本身能够引用服务器解析的系统属性。
- **InvokerURLPrefix**: 如果没有设置 InvokerURL，则将通过 InvokerURLPrefix + localhost + InvokerURLSuffix 组合构建它。比如，默认的 InvokerURLPrefix 取值为 “http://”。
- **InvokerURLSuffix**: 如果没有设置 InvokerURL，则将通过 InvokerURLPrefix + localhost + InvokerURLSuffix 组合构建它。比如，默认的 InvokerURLSuffix 取值为 “:8080/invoker/JMXInvokerServlet”。
- **UseHostName**: boolean 标志位，即应该使用 InetAddress.getHostName() 还是 getHostAddress() 方法的返回值作为上述 localhost 的值。如果为 true，则使用 getHostName()，否则使用 getHostAddress() 方法。
- **ExportedInterface**: 指定 HttpInvokerProxy 实现的 RMI 兼容接口名。

9. 借助于 HTTP 暴露任何 RMI 接口的步骤

使用 HttpProxyFactory MBean 和 JMX，并借助于 HTTP 作为传输协议，开发者能够将任何接口暴露出来供客户访问。虽然暴露的接口可以不是 RMI 接口，但它们必须兼容于 RMI，即所有的方法参数和返回值是可序列化的。同时，JBoss 对如下内容未提供支持，即被用于方法参数或返回值的 RMI 接口到其存根的转换。

借助于 HTTP 使得对象能够被调用，开发者需要完成以下 3 个步骤的内容。

步骤

(1) 使用 MarshalledInvocation.calculateHash 方法创建从 RMI 接口 Method 到 Long 的映射。比如，RMI SRPRemoteServerInterface 接口对应的过程如下：

```
import java.lang.reflect.Method;
import java.util.HashMap;
import org.jboss.invocation.MarshalledInvocation;
```

```

HashMap marshalledInvocationMapping = new HashMap();

// Build the Naming interface method map
Method[] methods = SRPRemoteServerInterface.class.getMethods();
for(int m = 0; m < methods.length; m++)
{
    Method method = methods[m];
    Long hash = new Long(MarshalledInvocation.calculateHash(method));
    marshalledInvocationMapping.put(hash, method);
}

```

(2) 创建或扩展现有的 MBean，以支持 invoke 操作。其方法签名为 Object invoke (Invocation invocation) throws Exception，如下给出的实例是用于接口 SRPRemoteServer Interface 的。它使用了步骤 1 中 marshalledInvocationMapping 含有的 Long 类型 Hash 对象，并将其映射到 SRPRemoteServerInterface 的 Method 中。

```

import org.jboss.invocation.Invocation;
import org.jboss.invocation.MarshalledInvocation;

public Object invoke(Invocation invocation) throws Exception
{
    SRPRemoteServerInterface theServer = <the_actual_rmi_server_object>;
    // Set the method hash to Method mapping
    if (invocation instanceof MarshalledInvocation)
    {
        MarshalledInvocation mi = (MarshalledInvocation) invocation;
        mi.setMethodMap(marshalledInvocationMapping);
    }
    // Invoke the Naming method via reflection
    Method method = invocation.getMethod();
    Object[] args = invocation.getArguments();
    Object value = null;
    try
    {
        value = method.invoke(theServer, args);
    }
    catch(InvocationTargetException e)
    {
        Throwable t = e.getTargetException();
        if(t instanceof Exception)
            throw (Exception) e;
        else
            throw new UndeclaredThrowableException(t, method.toString());
    }

    return value;
}

```

}

(3) 配置 HttpProxyFactory MBean，使得能够通过 JNDI 访问 RMI/HTTP 代理。比如：

```
<!-- Expose the SRP service interface via HTTP -->
<mbean code="org.jboss.invocation.http.server.HttpProxyFactory" name="jboss.security.tests:
service=SRP/HTTP">
  <attribute name="InvokerURL">http://localhost:8080/invoker/JMXInvokerServlet </attribute>
  <attribute name="InvokerName">jboss.security.tests:service=SRPService </attribute>
  <attribute name="ExportedInterface">
    org.jboss.security.srp.SRPRemoteServerInterface
  </attribute>
  <attribute name="JndiName">srp-test-http/SRPServiceInterface</attribute>
</mbean>
```

任何用户现在都能用 HttpProxyFactory（如 srp-test-http/SRPServiceInterface）指定的名字从 JNDI 中查找 RMI 接口，并同 RMI/JRMP 版一样获得代理。

第3章 JBoss 之命名——JNDI

命名服务

本章首先讨论基于 JNDI 的 JBoss 命名服务及 JNDI 在 JBoss 和 J2EE 中的作用。其次，本章还将介绍基本的 JNDI 应用编程接口和通用的使用约定。第三，本章将讨论由标准部署描述符定义的、JBoss 相关的 J2EE 组件命名环境配置。最后，本章还将讨论 JBoss 命名服务组件，即 Naming/JBoss 的配置和架构。

JBoss 命名服务是 Java 命名和目录接口（Java Naming and Directory Interface, JNDI）的实现。JNDI 在 J2EE 中担当了重要的作用，因为它提供的命名服务使得用户能够将名字映射到对象，这也是任何编程环境的基本需求。因为开发者和管理员需要通过可识别的名字获得对象和服务的引用。久负盛名的命名服务——Internet 域命名系统（DNS）就是这方面的典型。DNS 服务使得开发者通过逻辑名访问主机，而不是数字型的 Internet 地址。JNDI 在 J2EE 中担当了类似的角色，它使得开发者和管理员创建名字到对象的绑定，以供 J2EE 组件使用。

3.1 JNDI 概述

JNDI 是自 JDK 1.3 版本开始就绑定的标准 Java API。它为各种现有的命名服务提供了通用接口：DNS、LDAP、活动目录（译者注：Active Directory，即 AD）、RMI 注册器、COS 注册器、NIS 及文件系统。在逻辑上，JNDI API 被划分为：客户 API，供访问命名服务使用；服务供应商接口（Service Provider Interface, SPI），供用户创建命名服务的 JNDI 实现。

命名服务供应商必须实现抽象 SPI 层，使得客户能够使用核心 JNDI 类中的通用 JNDI 客户接口展示其命名服务。命名服务的 JNDI 实现称为 JNDI 供应商。JBoss 命名基于 SPI 类开发了 JNDI 实现。需要注意的是，J2EE 组件开发者不需要使用 JNDI SPI。

有关 JNDI 的全面介绍和教程，包括客户端和服务供应商 API，请开发者参考 Sun 教程，<http://java.sun.com/products/jndi/tutorial/>。

3.1.1 JNDI 应用编程接口

主要的 JNDI API 包为 `javax.naming`。其含有 5 个接口、10 个类及若干个异常。其中，存在 1 个关键类，`InitialContext`；2 个关键接口，`Context` 和 `Name`。

1. 名字

名字 (Name) 是 JNDI 的重要基础。命名系统决定了名字必须遵循的语法。命名系统的语法使得用户能够分析名字的字符串，并将其转换为组件。名字用于定位对象，简而言之，命名系统仅仅是具有不同名字的对象集合。为定位命名系统中的某对象，开发者需要提供名字，命名系统将返回该名字对应的对象。

比如，让我们考虑 UNIX 文件系统的命名规范。每个文件都是相对于文件系统根路径命名的，其中路径中的各个组件使用 “/” 分开。文件路径从左到右进行排列。比如，路径名 `/usr/jboss/readme.txt` 表示 `readme.txt` 是文件系统根路径中 `usr` 目录下的 `jboss` 目录中的文件。JBoss 命名使用 UNIX 风格的命名空间作为其命名规范。

`javax.naming.Name` 接口是通用名，即表示组件序列。它可能是合成名（由多个命名空间构成），或者复合名（用于单一的层次命名系统）。可以计算出名字中包含组件的数量。对于含有 N 个组件的名字而言，可以使用 0 到 $N-1$ 来表示，但不包括 N 。位于 0 位置的组件是最重要的组件。空值名没有组件。

合成名占据了多个命名空间的组件名排列。比如，通常用于 UNIX 命令 (`scp`) 的 `hostname+file` 就是合成名。再比如，利用 `scp` 命令将 `localfile.txt` 复制到主机 `ahost.someorg.org` 的 `tmp` 目录中，并将其名字改为 `remotefile.txt`：

```
scp localfile.txt ahost.someorg.org:/tmp/remotefile.txt
```

复合名来自层次命名空间。复合名中的各个组件都是原子名，即字符串不能够再细化为更小的组件。UNIX 文件系统中的文件路径名就是复合名。合成名，`ahost.someorg.org:/tmp/remotefile.txt` 占据了 DNS 和 UNIX 文件系统的命名空间，其包含的组件有 `ahost.someorg.org` 和 `/tmp/remotefile.txt`。组件是来自命名系统的命名空间中的字符串名。如果组件源于层次命名空间，则还可以借助于类 `javax.naming.CompoundName` 将其进一步细化，并获得各个原子组件。JNDI API 为合成名提供了 `Name` 接口的实现，即 `javax.naming.CompositeName` 类。

2. 上下文

`javax.naming.Context` 接口是与命名系统进行交互的主要接口。`Context` 接口表示名字到对象绑定的集合。每个上下文 (`Context`) 都有相关联的命名约定，即限定上下文如何将字符串名分析成 `javax.naming.Name` 实例。为创建名字到对象的绑定，开发者需要调用 `Context` 的 `bind` 方法，并给出名字和对象参数。然后，通过使用 `Context` 的 `lookup` 方法，开发者能够查找到名字对应的对象。通常，`Context` 提供了如下操作：将名字绑定到对象、释放绑定的名字及获得所有名字到对象的绑定列表。当然，绑定到 `Context` 的对象本身可以是 `Context` 类型。在开发者调用 `Context` 对象的 `bind` 方法时，被绑定的 `Context` 对象将作为其子上下文。

考虑 UNIX 文件系统上的上下文，比如某文件目录，其目录路径为 `/usr`。相对于其他文件目录命名的文件目录称为子上下文（通常也称为子目录）。路径名 `/usr/jboss` 的文件目录命名了 `jboss` 上下文，它是 `usr` 的子上下文。再比如，DNS 域中，`org` 为上下文。相对于其他 DNS 域命名的 DNS 域也是子上下文。对于 DNS 域 `jboss.org` 而言，由于解析 DNS 名的顺序是从右到左的，因此 `jboss` 是 `org` 的子上下文。

使用 InitialContext 获得上下文

所有命名服务操作都是借助于 Context 接口的具体实现完成的。因此，开发者需要为他们感兴趣的命名服务获得 Context。javax.naming.InitialContext 类实现了 Context 接口，并且它为实现与命名服务的交互提供了操作入口。

当创建 InitialContext 时，开发者通过环境属性能够实现其初始化工作。JNDI 是根据如下两个位置，并依次合并各个属性的取值：

- 来自构建器的环境参数中初次出现的属性、小应用程序参数和系统属性中的合适属性。
- 通过 classpath 寻找到的所有的 jndi.properties 资源文件。

对于那些在这两个位置都能寻找到的属性，其取值由如下规则确定：如果该属性为标准 JNDI 中规定的属性，比如指定 JNDI 工厂列表，则连接所有找到的属性值，并生成单个的、以冒号隔开的列表。对于其他属性，则仅仅使用初次发现的属性取值。当然，本书推荐使用 jndi.properties 文件指定 JNDI 环境属性。原因在于，将代码与 JNDI 具体供应商信息隔离开，如果日后需要更换 JNDI 供应商，则不用修改代码。最终，避免了再次编译代码。

具体的 Context 实现只有在运行时才能够确定。在默认情况下使用环境属性“java.naming.factory.initial”，其含有 javax.naming.spi.InitialContextFactory 实现的类名。通过命名服务供应商能够获得 InitialContextFactory 类名。

列表 3-1 为 jndi.properties 文件实例。其中，客户端应用将使用它连接运行于本地主机上 1099 端口的 JBossNS 服务。客户端应用将通过应用程序的 classpath 寻找 jndi.properties 文件。列表的内容都是 JBossNS JNDI 实现所要求的，其他 JNDI 供应商可能有不同属性及其取值。

列表 3-1 jndi.properties 文件实例

```
### JBossNS properties
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

3.1.2 J2EE 和 JNDI——应用组件环境

JNDI 是 J2EE 规范的基础部分，其主要用途在于将 J2EE 组件代码和其所部署的环境隔离开。使用应用组件环境能够实现应用组件的自定义，而不需要访问或改变现有的应用组件源代码。应用组件环境有时称之为企业命名上下文（ENC）。应用组件容器需要负责将 ENC 以 JNDI Context 的形式提供给容器组件。在 J2EE 组件生命周期中，ENC 可能的使用方式有：

（1）通过 ENC 访问应用组件业务逻辑以获得信息。组件供应商为组件使用标准部署描述符指定所需的 ENC 入口。在组件运行过程中，需要使用这些入口声明的信息和资源。

（2）容器提供工具，以允许组件部署者将组件开发者引用的 ENC 映射到部署环境中，从而满足引用要求。

(3) 组件部署者利用容器工具为最终部署准备组件。

(4) 在运行时, 组件容器使用部署包信息以创建完整的组件 ENC。

J2EE 平台中有关 JNDI 使用的完整规范可以参考 J2EE 1.3 规范。J2EE 规范位于 <http://java.sun.com/j2ee/download.html>。

应用组件实例使用 JNDI API 定位 ENC。它使用无参构建器创建 `javax.naming.InitialContext` 对象, 然后查找名字 `java:/comp/env` 下的命名环境。应用组件环境入口直接存放在 ENC 中, 或其子上下文中。列表 3-2 给出了组件访问 ENC 的代码原型。

列表 3-2 ENC 访问实例代码

```
// Obtain the application component's ENC
Context iniCtx = new InitialContext();
Context compEnv = (Context) iniCtx.lookup("java:comp/env");
```

应用组件环境是局部环境, 即当应用服务器容器控制线程和应用组件交互时, 它只能由该组件访问。其含义为, EJB Bean1 不能够访问 EJB Bean2 的 ENC 元素, 反之亦然。同理, Web 应用 Web1 不能够访问 Web 应用 Web2 或 Bean1、Bean2 的 ENC 元素。同时, 无论客户端是运行在应用服务器 JVM 中, 还是之外, 它都不能访问组件的 `java:comp` JNDI 上下文。ENC 的目的在于, 无论组件部署的环境类型, 它都能够提供应用组件所依赖的、隔离的只读命名空间。必须保证不同组件之间 ENC 的隔离性, 因为每个组件都定义了自身的 ENC 内容, 组件 A 和 B 有可能定义了相同的名字指向不同对象。比如, EJB Bean1 可能定义了环境入口 `java:/comp/env/red` 指向十六进制值, 以用于 RGB 颜色中的红色。而 Web 应用 Web1 将相同的名字绑定到部署环境中红色的本地化表示。

通常情况下, 在 JBossNS 实现中, 一共使用了 3 种命名范围: `java:comp` 下的名字、`java:` 下的名字及其他名字。其中, `java:comp` 上下文及其子上下文仅对关联 `java:comp` 上下文的应用组件可用。`java:` 上下文及其子上下文仅对 JBoss 服务器虚拟机可见。其他任何上下文或对象绑定适用于远程客户, 其提供的上下文或对象支持序列化。“JBossNS 架构”节内容给出了如何隔离这些命名范围的介绍。

比如, `javax.sql.DataSource` 连接工厂仅仅能在对应数据库池驻留的 JBoss 虚拟机中可用, 它是绑定到 `java:` 上下文的。再比如, EJB home 接口是全局可见名, 即允许远程客户访问。

1. ENC 使用约定

JNDI API 能够在外部配置应用组件的许多信息。应用组件使用 JNDI 的名字访问 EJB 组件中标准 `ejb-jar.xml` 配置描述符, 或 Web 组件中标准 `web.xml` 部署描述符声明的信息。通过 JNDI 存储和获得的信息类型有如下几种。

- 由 `env-entry` 元素声明的环境入口。
- 由 `ejb-ref` 和 `ejb-local-ref` 元素声明的 EJB 引用。
- 由 `resource-ref` 元素声明的资源管理器连接工厂引用。
- 由 `resource-env-ref` 元素声明的资源环境引用。

各种部署描述符元素都存在于 JNDI 的使用约定中, 即信息绑定的 JNDI 上下文名字。同时, 除了标准部署描述符元素外, 还有 JBoss 服务器相关的部署描述符元素, 它将应用

组件使用的 JNDI 名映射到部署环境 JNDI 名中。

(1) ejb-jar.xml ENC 元素

EJB 2.0 部署描述符包括 EJB 组件及其环境集合。3 种类型的 EJB 组件：会话、实体以及消息驱动 Bean，它们都支持使用 EJB 局部命名上下文。ejb-jar.xml 描述符提供了 EJB 操作环境的逻辑视图。由于 EJB 组件开发者通常都不知道部署 EJB 的目标环境，因此开发者使用逻辑名字，以独立于部署环境的方式给出组件环境。部署管理者需要将 EJB 组件逻辑名字和对应的部署环境资源联系起来。

图 3-1 给出了 EJB 部署描述符 DTD 中 ENC 元素部分的图形化视图。由于实体与消息驱动 Bean 类似，因此只是展开了会话 Bean 的 ENC 元素。通过 Sun 网站能够找到完整的 ejb-jar.xml DTD。

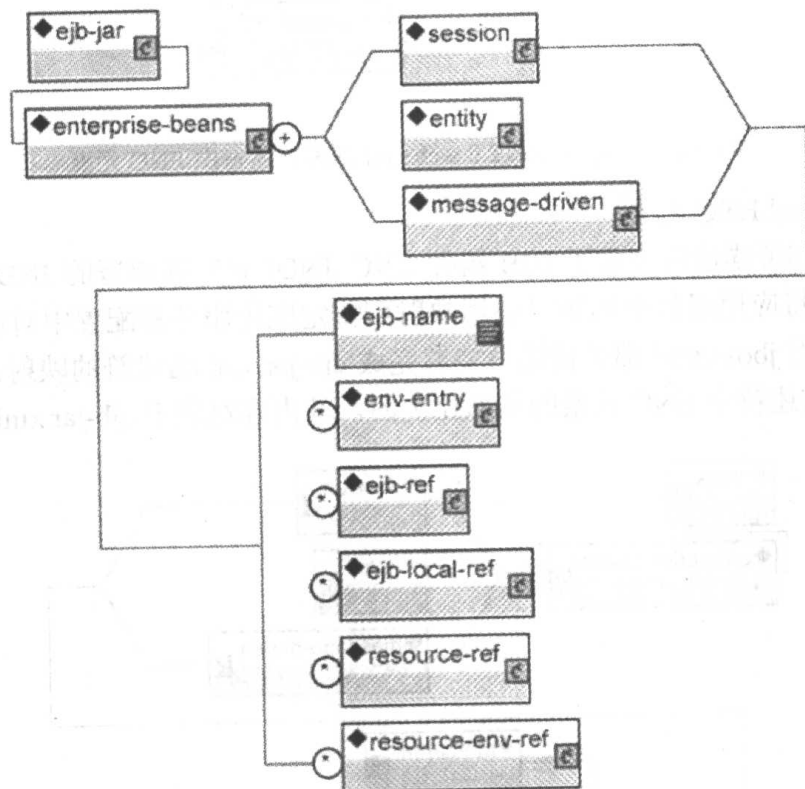


图 3-1 标准 ejb-jar.xml 2.0 部署描述符中的 ENC 元素

(2) web.xml ENC 元素

Servlet 2.3 部署描述符包括 Web 组件及其环境集合。Web 应用中声明的 ENC 可以供其中所有的 Servlet 和 JSP 页面使用。由于 Web 应用开发者通常不知道 Web 应用的目标部署环境，因此开发者必须使用逻辑名字，以独立于部署环境的方式给出组件环境。部署管理者需要将 Web 组件逻辑名字连接到相应的部署环境资源中。

图 3-2 给出了 Web 应用部署描述符 DTD 中 ENC 元素的图形化视图。通过 Sun 网站 http://java.sun.com/dtd/web-app_2_3.dtd，开发者可以获得完整的 web.xml DTD。

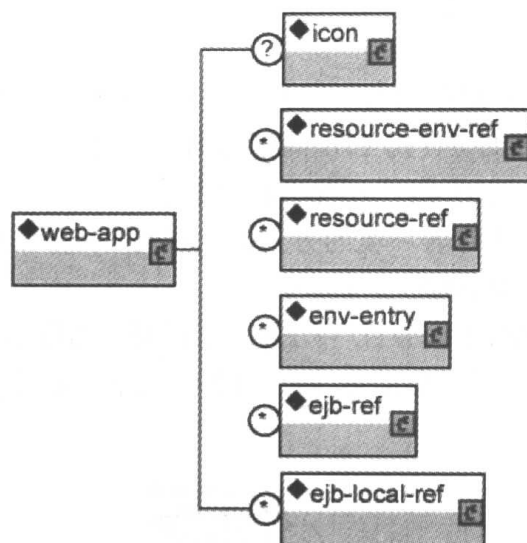


图 3-2 标准 Servlet 2.3 web.xml 部署描述符中 ENC 元素

(3) jboss.xml ENC 元素

JBoss EJB 部署描述符完成了 EJB 组件 ENC JNDI 到实际部署的 JNDI 名字的映射。应用开发者需要将应用组件中的逻辑引用映射到特定应用服务器配置中对应的物理资源。在 JBoss 中，使用 jboss.xml 部署描述符能够完成 ejb-jar.xml 描述符的映射。图 3-3 给出了 JBoss EJB 部署描述符中 ENC 元素的图形化视图，其内容对应于 ejb-jar.xml 的各个元素。

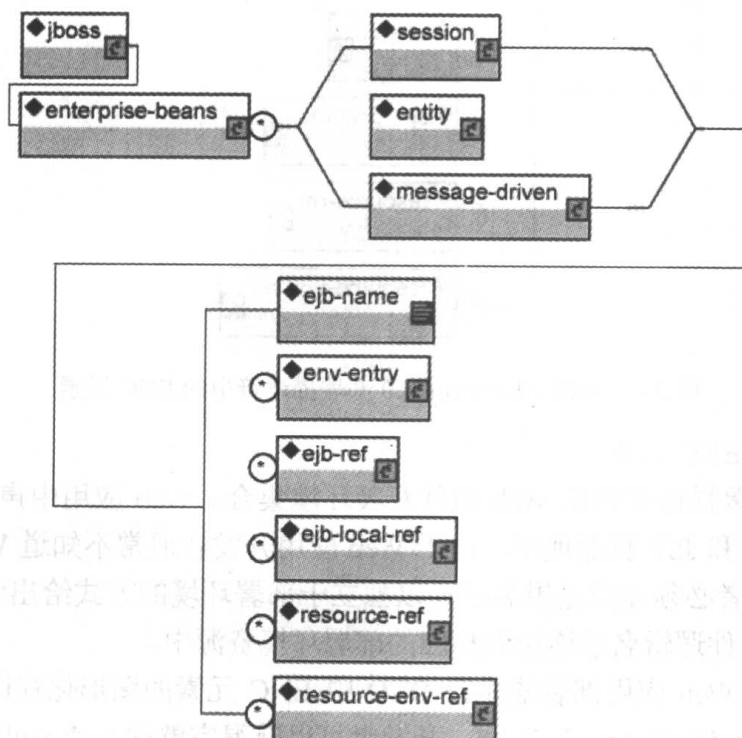


图 3-3 JBoss 3.2 jboss.xml 部署描述符中的 ENC 元素

(4) jboss-web.xml ENC 元素

JBoss Web 部署描述符将 Web 应用 ENC JNDI 名字映射到实际部署的 JNDI 名字上。应用开发者需要将 Web 应用中的逻辑引用映射到特定应用服务器配置中对应的物理资源。在 JBoss 中, 使用 jboss-web.xml 部署描述符能够完成 web.xml 描述符的映射。图 3-4 给出了 JBoss Web 部署描述符 DTD 中 ENC 元素的图形化视图。通过 JBoss 网站 http://www.jboss.org/j2ee/dtd/jboss_web_3_2.dtd, 或者发布版的 docs/dtd 目录, 开发者能够找到完整的 jboss-web.xml DTD 文件。

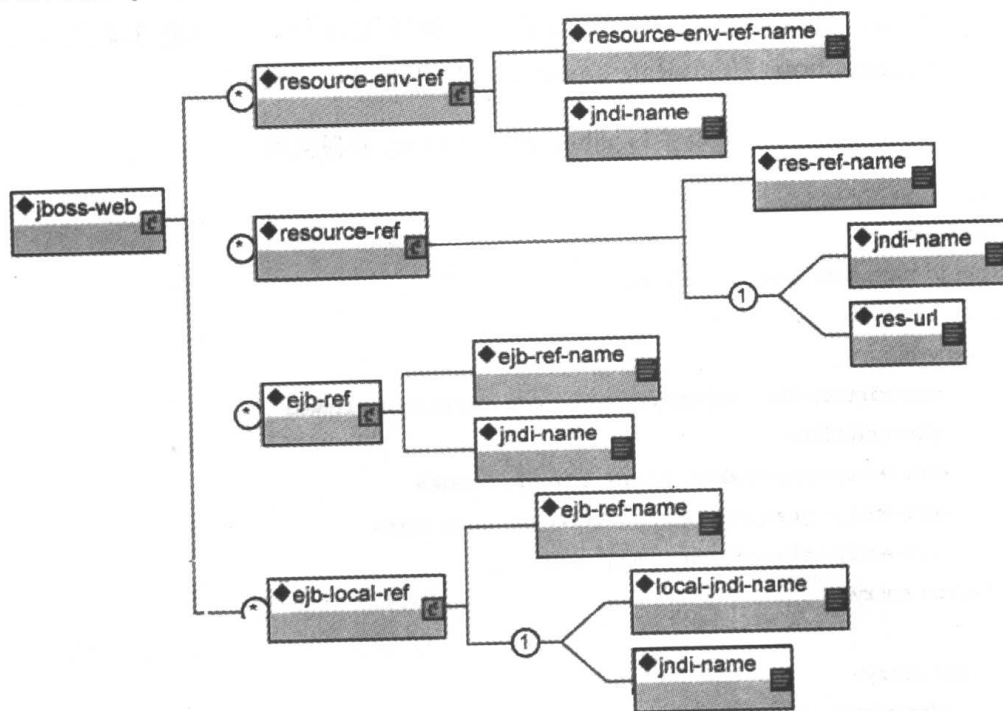


图 3-4 JBoss 3.2 jboss-web.xml 部署描述符中的 ENC 元素

2. 环境入口

环境入口 (environment entry) 是组件 ENC 中存储信息的最简单方法, 它类似于操作系统的环境变量, 比如 UNIX 或 Windows。环境入口是名字 - 值的绑定, 即允许组件外部配置其取值和使用名字引用该值。

通过标准部署描述符中的 env-entry 元素能够声明环境入口。env-entry 元素包含如下子元素。

- 可选 description 元素, 提供入口的描述。
- env-entry-name 元素, 给出相对于 java:comp/env 的入口名字。
- env-entry-type 元素, 指定入口值的 Java 类型, 其取值必须是以下列举的其中之一:
 - java.lang.Byte
 - java.lang.Boolean
 - java.lang.Character
 - java.lang.Double

- java.lang.Float
- java.lang.Integer
- java.lang.Long
- java.lang.Short
- java.lang.String

- env-entry-value 元素, 通过字符串形式给出入口取值。

列表 3-3 给出了 ejb-jar.xml 部署描述符中 env-entry 的实例片段。由于 env-entry 是完整的名字和值规范, 因此不存在 JBoss 相关的部署描述符元素。列表 3-4 给出了访问列表 3-3 声明的 maxExemptions 和 taxRate env-entry 值的代码片段实例。

列表 3-3 ejb-jar.xml env-entry 片段实例

```
...
<session>
  <ejb-name>ASessionBean</ejb-name>
  ...
  <env-entry>
    <description>The maximum number of tax exemptions allowed
    </description>
    <env-entry-name>maxExemptions</env-entry-name>
    <env-entry-type>java.lang.Integer</env-entry-type>
    <env-entry-value>15</env-entry-value>
  </env-entry>

  <env-entry>
    <description>The tax rate
    </description>
    <env-entry-name>taxRate</env-entry-name>
    <env-entry-type>java.lang.Float</env-entry-type>
    <env-entry-value>0.23</env-entry-value>
  </env-entry>
</session>
...
```

列表 3-4 ENC env-entry 访问代码片段

```
InitialContext iniCtx = new InitialContext();
Context envCtx = (Context) iniCtx.lookup("java:comp/env");
Integer maxExemptions = (Integer) envCtx.lookup("maxExemptions");
Float taxRate = (Float) envCtx.lookup("taxRate");
```

3. EJB 引用

EJB 和 Web 组件经常需要和其他 EJB 组件交互。由于 EJB home 接口绑定的 JNDI 名属于部署时的行为, 因此需要一种机制, 告知组件开发者声明对 EJB 的引用, 使得部署者能够连接到该 EJB。EJB 引用能够满足该需求。

EJB 引用是应用组件命名环境中指向已部署 EJB home 接口的连接。应用组件使用的名字是逻辑连接,即将组件与其在部署环境中的实际 EJB home 名字隔离开。J2EE 规范建议开发者,所有对 EJB 的引用都组织在应用组件环境的 `java:comp/env/ejb` 上下文中。

使用部署描述符中的 `ejb-ref` 元素能够声明 EJB 引用。每个 `ejb-ref` 元素给出了接口需求,即引用应用组件对被引用企业 Bean 的接口需求。`ejb-ref` 元素含有如下子元素。

- 可选的 `description` 元素,给出引用的目的。
- `ejb-ref-name` 元素,指定相对于 `java:comp/env` 上下文的引用名,建议将引用放置在 `java:comp/env/ejb` 上下文的下面,并使用 `ejb/linkname` 形式给出 `ejb-ref-name` 值。
- `ejb-ref-type` 元素,指定 EJB 类型。它或者是 `Entity`,或者是 `Session`。
- `home` 元素,给出了 EJB home 接口的全限定类名。
- `remote` 元素,给出了 EJB 远程接口的全限定类名。
- 可选的 `ejb-link` 元素,将引用连接到其他 EJB,它位于 `ejb-jar` 文件或同一 J2EE 应用单元中。`ejb-link` 值是被引用企业 Bean 的 `ejb-name`。如果多个企业 Bean 的 `ejb-name` 相同,则使用路径名以指定包含被引用组件的 `ejb-jar` 文件。其中,该路径名相对于引用 `ejb-jar` 文件的路径名。应用组装者将被引用企业 Bean 的 `ejb-name` 追加到路径名,并以“#”隔开。从而,能够惟一标识出具有相同名字的企业 Bean。

EJB 引用的作用范围仅仅在声明了 `ejb-ref` 元素的应用组件内。其含义为在运行时,不允许其他应用组件访问该应用组件的 EJB 引用。同时,其他应用组件定义的 `ejb-ref` 元素也可能包含相同的 `ejb-ref-name` 值,但不会引起名字冲突。列表 3-5 给出了 `ejb-jar` 中使用 `ejb-ref` 元素的片段。列表 3-6 给出了实例代码,以访问列表 3-5 声明的 `ShoppingCartHome` 引用。

列表 3-5 `ejb-jar.xml` 中 `ejb-ref` 描述符实例片段

```
...
<session>
    <ejb-name>ShoppingCartBean</ejb-name>
    ...
</session>

<session>
    <ejb-name>ProductBeanUser</ejb-name>
    ...
    <ejb-ref>
        <description>This is a reference to the store products entity</description>
        <ejb-ref-name>ejb/ProductHome</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <home>org.jboss.store.ejb.ProductHome</home>
    </ejb-ref>
    <remote> org.jboss.store.ejb.Product</remote>
</session>
```

```
<session>
  <ejb-ref>
    <ejb-name>ShoppingCartUser</ejb-name>
    ...
    <ejb-ref-name>ejb/ShoppingCartHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>org.jboss.store.ejb.ShoppingCartHome</home>
    <remote> org.jboss.store.ejb.ShoppingCart</remote>
    <ejb-link>ShoppingCartBean</ejb-link>
  </ejb-ref>
</session>

<entity>
  <description>The Product entity bean</description>
  <ejb-name>ProductBean</ejb-name>
  ...
</entity>
...
```

列表 3-6 ENC ejb-ref 访问代码片段

```
InitialContext iniCtx = new InitialContext();
Context ejbCtx = (Context) iniCtx.lookup("java:comp/env/ejb");
ShoppingCartHome home = (ShoppingCartHome) ejbCtx.lookup("ShoppingCartHome");
```

（1）EJB 引用（jboss.xml 和 jboss-web.xml）

JBoss 服务器的 jboss.xml EJB 部署描述符通过两种方式能够影响 EJB 引用。

其一，session 和 entity 元素的 jndi-name 子元素允许用户指定 EJB home 接口的 JNDI 名字。如果 jboss.xml 中的 jndi-name 不存在，则 home 接口将绑定到 ejb-jar.xml 中的 ejb-name 值上。比如，如果 jboss.xml jndi-name 不存在，则列表 3-5 中 ejb-name 为 ShoppingCartBean 的会话 Bean 将会把 home 接口绑定在 ShoppingCartBean 上。

其二，组件 ENC ejb-ref 引用的目的地。ejb-link 元素不能够引用其他企业应用中的 EJB。如果 ejb-ref 需要访问外部 EJB，则可以使用 jboss.xml 中的 ejb-ref/jndi-name 元素指定已部署 EJB 的 JNDI 名字。

jboss-web.xml 描述符仅仅用于设置 Web 应用 ENC ejb-ref 引用的目的 EJB。JBoss ejb-ref 的内容模型如下。

- ejb-ref-name 元素，对应于 ejb-jar.xml 或 web.xml 标准描述符中的 ejb-ref-name 元素。
- jndi-name 元素，用于指定部署环境中 EJB home 接口的 JNDI 名字。

列表 3-7 所给出的 jboss.xml 描述符实例片段重点强调了如下要点：

- ProductBeanUser ejb-ref 所连接的目的地设置为部署名 jboss/store/ProductHome。
- ProductBean 的部署 JNDI 名字设置为 jboss/store/ProductHome。

列表 3-7 jboss.xml ejb-ref 实例片段

```
...
<session>
  <ejb-name>ProductBeanUser</ejb-name>
  <ejb-ref>
    <ejb-ref-name>ejb/ProductHome</ejb-ref-name>
    <jndi-name>jboss/store/ProductHome</jndi-name>
  </ejb-ref>
</session>

<entity>
  <ejb-name>ProductBean</ejb-name>
  <jndi-name>jboss/store/ProductHome</jndi-name>
  ...
</entity>
...
```

(2) EJB 本地引用

EJB 2.0 引入了非远程接口，即本地接口。本地接口不使用 RMI 的调用传值语义，而是使用调用传址语义，从而不会出现任何 RMI 序列化行为。EJB 本地引用实现了应用组件命名约定中指向已部署 EJB 本地 home 接口的连接。应用组件使用的名字是逻辑连接，即将组件与其在部署环境中的实际 EJB 本地 home 名字隔离开。J2EE 规范建议所有对 EJB 的引用都组织在应用组件环境的 `java:comp/env/ejb` 上下文中。

开发者使用配置描述符中的 `ejb-local-ref` 元素能够声明 EJB 本地引用。每个 `ejb-local-ref` 元素都给出了接口需求，即引用应用组件对被引用企业 Bean 的接口需求。`ejb-local-ref` 元素含有如下子元素。

- 可选的 `description` 元素，给出了引用的目的。
- `ejb-ref-name` 元素，指定相对于 `java:comp/env` 上下文的引用名。建议开发者将引用放置在 `java:comp/env/ejb` 上下文的下面，并使用 `ejb/linkname` 形式给出 `ejb-ref-name` 值。
- `ejb-ref-type` 元素，指定 EJB 类型。它或者是 `Entity`，或者是 `Session`。
- `local-home` 元素，给出 EJB 本地 home 接口的全限定类名。
- `local` 元素，给出 EJB 本地接口的全限定类名。
- 可选的 `ejb-link` 元素，将引用连接到其他 EJB，它位于 `ejb-jar` 文件或同一 J2EE 应用单元中。`ejb-link` 值是被引用企业 Bean 的 `ejb-name`。如果多个企业 Bean 的 `ejb-name` 相同，则使用路径名以指定包含被引用组件的 `ejb-jar` 文件。其中，该路径名相对于引用 `ejb-jar` 文件的路径名。应用组装者将被引用企业 Bean 的 `ejb-name` 追加到路径名，并用“#”隔开。从而，能够惟一标识具有相同名字的多个企业 Bean。另外，在 JBoss 中，必须指定 `ejb-link` 元素以匹配相应 EJB 的本地引用。

EJB 本地引用的作用范围仅仅在声明了 `ejb-local-ref` 元素的应用组件内。其含义为：在运行时，不允许其他应用组件访问该应用组件的 EJB 本地引用。同时，其他应用组件定义的 `ejb-local-ref` 元素也可能包含相同的 `ejb-ref-name` 值，但不会引起名字冲突。列表 3-8 给出了 `ejb-jar` 中使用 `ejb-local-ref` 元素的片段。列表 3-9 给出了实例代码，以访问列表 3-8 声明的 `ProbeLocalHome` 引用。

列表 3-8 `ejb-jar.xml` 中 `ejb-local-ref` 元素实例片段

```
...
<session>
  <ejb-name>Probe</ejb-name>
  <home>org.jboss.test.perf.interfaces.ProbeHome</home>
  <remote>org.jboss.test.perf.interfaces.Probe</remote>
  <local-home>org.jboss.test.perf.interfaces.ProbeLocalHome</localhome>
  <local>org.jboss.test.perf.interfaces.ProbeLocal</local>
  <ejb-class>org.jboss.test.perf.ejb.ProbeBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Bean</transaction-type>
</session>

<session>
  <ejb-name>PerfTestSession</ejb-name>
  <home>org.jboss.test.perf.interfaces.PerfTestSessionHome</home>
  <remote>org.jboss.test.perf.interfaces.PerfTestSession</remote>
  <ejb-class>org.jboss.test.perf.ejb.PerfTestSessionBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>
  <ejb-ref>
    <ejb-ref-name>ejb/ProbeHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>org.jboss.test.perf.interfaces.SessionHome</home>
    <remote>org.jboss.test.perf.interfaces.Session</remote>
    <ejb-link>Probe</ejb-link>
  </ejb-ref>
  <ejb-local-ref>
    <ejb-ref-name>ejb/ProbeLocalHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>org.jboss.test.perf.interfaces.ProbeLocalHome</localhome>
    <local>org.jboss.test.perf.interfaces.ProbeLocal</local>
    <ejb-link>Probe</ejb-link>
  </ejb-local-ref>
</session>
...
```

列表 3-9 ENC `ejb-local-ref` 访问代码片段

```
InitialContext iniCtx = new InitialContext();
```



```
Context ejbCtx = (Context) iniCtx.lookup("java:comp/env/ejb");
ProbeLocalHome home = (ProbeLocalHome) ejbCtx.lookup("ProbeLocalHome");
```

(3) 资源管理器连接工厂引用

资源管理器连接工厂引用允许应用组件代码使用逻辑名字引用资源工厂。它由标准部署描述符中的 `resource-ref` 元素定义。部署者使用 `jboss.xml` 和 `jboss-web.xml` 描述符将资源管理器工厂引用绑定到目标实际操作环境中的资源管理器连接工厂上。

每个 `resource-ref` 元素指定单个资源管理器连接工厂引用。`resource-ref` 元素由如下子元素构成。

- 可选的 `description` 元素，给出了引用的目的。
- `res-ref-name` 元素，指定相对于 `java:comp/env` 上下文的引用名。至于不同资源类型的命名约定在后续有介绍。
- `res-type` 元素，指定资源管理器连接工厂的全限定类名。
- `res-auth` 元素，给出资源的登录操作是否由应用组件代码完成，还是由容器基于部署者提供的 `Principal` 映射信息自动完成。其取值是 `Application`，或者是 `Container`。
- 可选的 `res-sharing-scope` 元素。当前 JBoss 并不提供支持。

J2EE 规范建议，将所有的资源管理器连接工厂引用组织在应用组件环境的子上下文的下面，对于每个资源管理器类型使用不同的子上下文。建议为子上下文名提供的资源管理器类型如下。

- JDBC `DataSource` 引用应该声明在 `java:comp/env/jdbc` 子上下文中。
- JMS 连接工厂应该声明在 `java:comp/env/jms` 子上下文中。
- JavaMail 连接工厂应该声明在 `java:comp/env/mail` 子上下文中。
- URL 连接工厂应该声明在 `java:comp/env/url` 子上下文中。

列表 3-10 给出了 `web.xml` 描述符片段，以解释 `resource-ref` 的元素用法。列表 3-11 给出了应用组件代码片段，以访问列表 3-10 定义的 `DefaultMail` 资源。

列表 3-10 web.xml 中 resource-ref 元素片段

```
<web>
...
  <servlet>
    <servlet-name>AServlet</servlet-name>
    ...
  </servlet>
  ...
  <!-- JDBC DataSources (java:comp/env/jdbc) -->
  <resource-ref>
    <description>The default DS</description>
    <res-ref-name>jdbc/DefaultDS</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
```

```

<!-- JavaMail Connection Factories (java:comp/env/mail) -->
<resource-ref>
  <description>Default Mail</description>
  <res-ref-name>mail/DefaultMail</res-ref-name>
  <res-type>javax.mail.Session</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
<!-- JMS Connection Factories (java:comp/env/jms) -->
<resource-ref>
  <description>Default QueueFactory</description>
  <res-ref-name>jms/QueueFactory</res-ref-name>
  <res-type>javax.jms.QueueConnectionFactory</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
</web>

```

列表 3-11 ENC resource-ref 访问实例代码片段

```

Context initCtx = new InitialContext();
javax.mail.Session s = (javax.mail.Session)
initCtx.lookup("java:comp/env/mail/DefaultMail");

```

(4) 资源管理器连接工厂引用 (jboss.xml 和 jboss-web.xml)

JBoss jboss.xml EJB 部署描述符和 jboss-web.xml Web 应用部署描述符为 res-ref-name 元素定义的逻辑名连接到部署在 JBoss 上资源工厂的 JNDI 名提供服务。通过为 jboss.xml 或 jboss-web.xml 描述符提供 resource-ref 元素能实现上述目的。resource-ref 元素由下列子元素组成。

- res-ref-name 元素, 它必须匹配 ejb-jar.xml 或 web.xml 标准描述符中 resource-ref 元素对应的 res-ref-name。
- 可选的 res-type 元素, 指定资源管理器连接工厂的全限定类名。
- jndi-name 元素, 指定部署在 JBoss 上资源工厂的 JNDI 名。
- res-url 元素, 如果 resource-ref 类型为 java.net.URL, 则需要为该元素指定 URL 字符串。

列表 3-12 给出的 jboss-web.xml 描述符实例, 展示了列表 3-10 中 resource-ref 元素的映射实例。

列表 3-12 jboss-web.xml 中的 resource-ref 实例片段

```

<jboss-web>
...
  <resource-ref>
    <res-ref-name>jdbc/DefaultDS</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <jndi-name>java:/DefaultDS</jndi-name>
  </resource-ref>
  <resource-ref>

```

```

    <res-ref-name>mail/DefaultMail</res-ref-name>
    <res-type>javax.mail.Session</res-type>
    <jndi-name>java:/Mail</jndi-name>
  </resource-ref>
  <resource-ref>
    <res-ref-name>jms/QueueFactory</res-ref-name>
    <res-type>javax.jms.QueueConnectionFactory</res-type>
    <jndi-name>QueueConnectionFactory</jndi-name>
  </resource-ref>
  ...
</jboss-web>

```

4. 资源环境引用

资源环境引用元素通过逻辑名引用受管对象，该对象和资源相关（比如，JMS 目的地）。资源环境引用是通过标准部署描述符中的 `resource-env-ref` 元素定义的。部署者使用 `jboss.xml` 和 `jboss-web.xml` 描述符将资源环境引用绑定到目标实际操作环境中的受管对象上。

每个 `resource-env-ref` 元素给出了引用应用组件对被引用受管对象的需求信息。`resource-env-ref` 元素由下列子元素构成。

- 可选的 `description` 元素，给出了引用的目的。
- `resource-env-ref-name` 元素，指定相对于 `java:comp/env` 上下文的引用名。该子上下文放置的名字必须和对应的资源工厂类型一致。比如，名字为 `MyQueue` 的 JMS queue 引用应该提供值为 `jms/MyQueue` 的 `resource-env-ref-name`。
- `resource-env-ref-type` 元素，指定被引用对象的全限定类名。比如，对于 JMS queue，则该值为 `javax.jms.Queue`。

列表 3-13 给出了某会话 Bean 声明的 `resource-ref-env` 元素实例。列表 3-14 给出了应用客户端，以使用列表 3-13 给出的 `resource-ref-env` 元素实例。

列表 3-13 ejb-jar.xml 中 resource-env-ref 实例片段

```

<session>
  <ejb-name>MyBean</ejb-name>
  ...
  <resource-env-ref>
    <description>This is a reference to a JMS queue used in the
      processing of Stock info
    </description>
    <resource-env-ref-name>jms/StockInfo</resource-env-ref-name>
    <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
  </resource-env-ref>
  ...
</session>

```

列表 3-14 ENC resource-env-ref 访问代码片段

```
InitialContext iniCtx = new InitialContext();
javax.jms.Queue q = (javax.jms.Queue)
    envCtx.lookup("java:comp/env/jms/StockInfo");
```

资源环境引用（jboss.xml 和 jboss-web.xml）

JBoss jboss.xml EJB 部署描述符和 jboss-web.xml Web 应用部署描述符为 resource-env-ref-name 元素定义的逻辑名连接到部署在 JBoss 上受管对象的 JNDI 名提供服务。通过 jboss.xml 或 jboss-web.xml 描述符的 resource-env-ref 元素能够实现上述目的。resource-env-ref 元素由下列子元素构成。

- resource-env-ref-name 元素，它必须匹配 ejb-jar.xml 或 web.xml 标准描述符中 resource-env-ref 元素对应的 resource-env-ref-name 值。
- jndi-name 元素，指定部署于 JBoss 上的资源的 JNDI 名。

列表 3-15 提供了 jboss.xml 描述符实例片段，展示了列表 3-13 中 resource-env-ref 元素的映射实例。

列表 3-15 jboss.xml 中 resource-env-ref 实例片段

```
<session>
  <ejb-name>MyBean</ejb-name>
  ...
  <resource-env-ref>
    <resource-env-ref-name>jms/StockInfo</resource-env-ref-name>
    <jndi-name>queue/StockInfoQue</jndi-name>
  </resource-env-ref>
  ...
</session>
```

3.2 JBossNS 架构

JBossNS 架构是基于 Java 套接字/RMI 的，并实现了 javax.naming.Context 接口。它是客户端/服务器端模式实现，并允许远程访问。经过优化的 JBossNS 实现，使得从运行 JBossNS 服务器的 JVM 中访问它时不需要涉及到套接字。在同一 JVM 中通过对象引用能够实现访问的目的。图 3-5 给出了 JBossNS 实现中的若干个关键类及其关系。

本节先介绍 NamingService MBean。NamingService MBean 提供了 JNDI 命名服务，它是 J2EE 技术组件普遍使用的关键服务。NamingService 的配置属性如下。

- **Port:** 用于 NamingService 的 JNP 协议监听端口。其默认值为 1099，如果没有指定，则其值和 RMI 注册器的默认端口相同。
- **RmiPort:** RMI 命名实现导出的 RMI 端口。如果没有指定，则为 0，即可以使用任何可用端口。
- **BindAddress:** NamingService 监听的具体地址。它能够用于存在多个主机地址的

机器上，即为 `java.net.ServerSocket` 提供监听地址。但只有其中一个地址接受客户请求。

- **RmiBindAddress:** NamingService 监听 RMI 服务器部分的具体地址。它能够用于存在多个主机地址的机器上, 即为 `java.net.ServerSocket` 提供监听地址。但只有其中一个地址能够接受客户请求。如果开发者没有指定 `RmiBindAddress` 值, 则 `RmiBindAddress` 默认情况下设置为 `BindAddress` 值。
- **Backlog:** 所允许连接请求的最大队列长度。如果队列已满且有连接请求时, 该请求将被拒绝。
- **ClientSocketFactory:** 可选的自定义 `java.rmi.server.RMIClientSocketFactory` 实现的类名。如果没有指定, 则使用默认值 `RMIClientSocketFactory`。
- **ServerSocketFactory:** 可选的自定义 `java.rmi.server.RMIServerSocketFactory` 实现的类名。如果没有指定, 则使用默认值 `RMIServerSocketFactory`。
- **JNPServerSocketFactory:** 可选的自定义 `javax.net.ServerSocketFactory` 实现的类名。它是 `ServerSocket` 的工厂, 用于引导 JBossNS Naming 接口的下载。如果没有指定, 则使用 `javax.net.ServerSocketFactory.getDefault()` 方法的返回值。

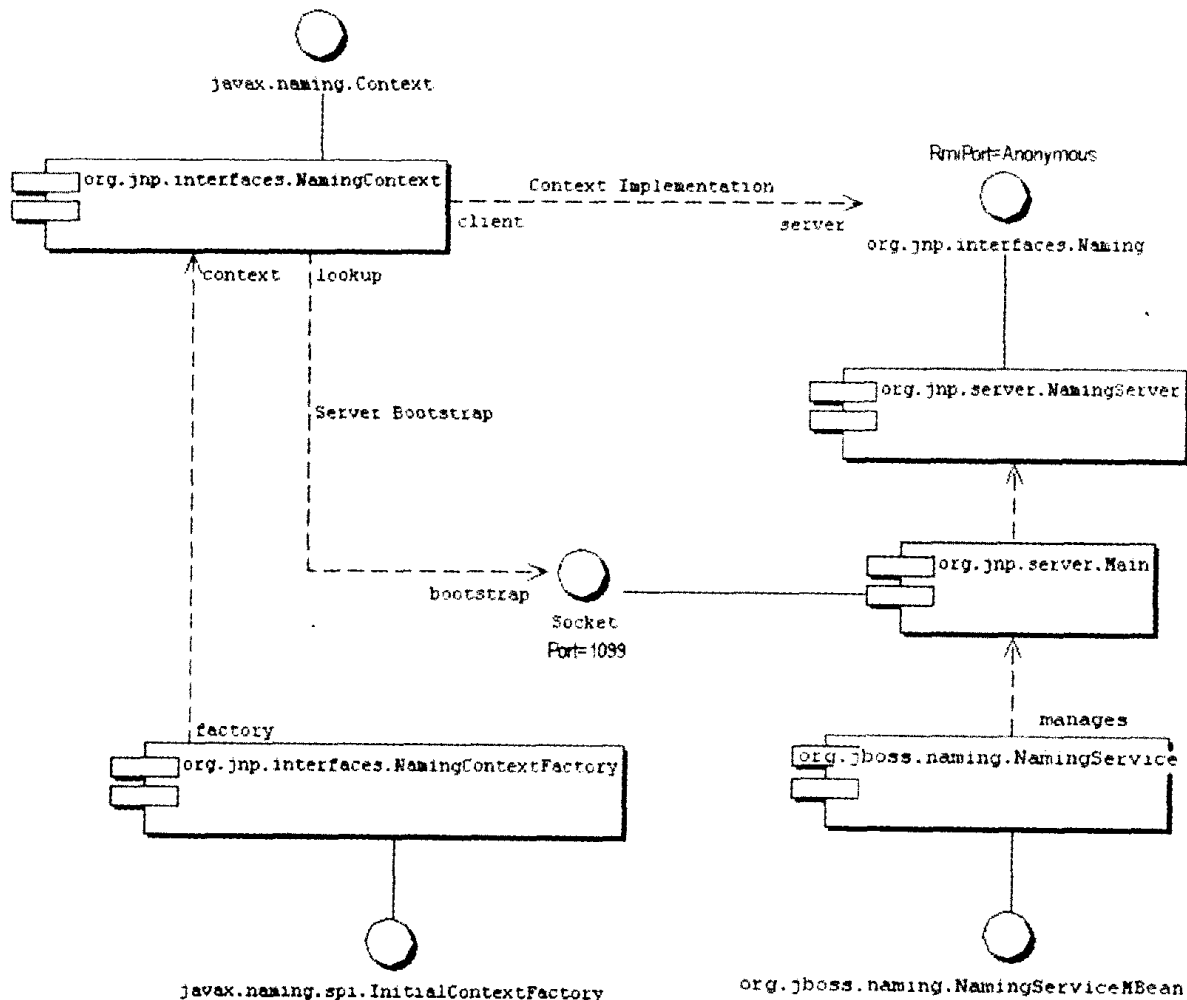


图 3-5 JBossNS 架构中的关键类

NamingService 也创建了 java:comp 上下文，有权访问 java:comp 上下文线程的上下文 ClassLoader 能有效隔离该 java:comp 上下文。因此，满足了 J2EE 规范要求，即为应用组件提供私有 ENC。借助于 ClassLoader 隔离 java:comp 上下文是通过将 javax.naming.Reference 绑定到 Context 上实现的。其中，上述过程使用到的 javax.naming.ObjectFactory 接口实现是 org.jboss.naming.ENCFactory 类。当客户查找 java:comp 或其子上下文时，ENCFactory 会检查线程的上下文类装载器，并使用该类装载器作为 map 的键以完成查找工作。如果没有找到匹配 ClassLoader 的 Context 实例，则会创建一个实例，并将创建的 Context 实例放置在 ENCFactory 的 map 中以关联该 Classloader。因此，能否正确隔离应用组件的 ENC 完全取决于各个组件能够接收到与组件的运行线程关联的惟一 ClassLoader。NamingService 将其功能委派给 org.jnp.server.Main MBean。引入该 MBean 的原因在于，JBossNS 起初是能够独立运行的 JNDI 实现，当然也可以以现在这种方式运行。NamingService MBean 将 Main 实例嵌入在 JBoss 服务器中，使得同 JBoss 服务器使用同一 JVM 的 JNDI 不用涉及套接字访问。NamingService MBean 实际上是使用了 JBossNS Main MBean 的配置属性。设置 NamingService MBean 的任何属性也就完成了其包含的 Main MBean 对应属性的设置。当 NamingService 启动后，它将启动其包含的 Main MBean，从而能够激活 JNDI 命名服务。另外，NamingService 通过 JMX 提供的弱类型化（detyped）Invoke 操作暴露 Naming 接口操作，这使得基于任何协议实现的 JMX 适配器能够实现对命名服务的访问。本章后续内容将会演示，借助于 Invoke 操作如何使用 HTTP 访问命名服务。

本书没有给出线程及其上下文类装载器的具体细节介绍，但 JNDI 教程给出了较完整的介绍。具体介绍在 <http://java.sun.com/products/jndi/tutorial/beyond/misc/classloader.html> 中。

当 Main MBean 启动时，将完成如下任务。

- 初始化 org.jnp.naming.NamingServer 实例，并将其设置成本地 JVM 服务器中的实例。在运行 JBoss 的 JVM 中创建的 org.jnp.interfaces.NamingContext 实例将使用上述 NamingServer 实例。因此，避免了通过 TCP/IP 对 RMI 的调用。
- 使用配置好的 RmiPort、ClientSocketFactory 及 ServerSocketFactory 属性导出 NamingService 实例的 RMI 接口，即 org.jnp.naming.interfaces.Naming。
- 创建 Socket，以监听 BindAddress 和 Port 属性给定的接口。
- 创建线程以接受 Socket 连接请求。

3.2.1 命名 InitialContext 工厂

当前，JBoss JNDI 供应商支持 3 种不同的 InitialContext 工厂实现。其中，使用最多的工厂实现是 org.jnp.interfaces.NamingContextFactory。其包括的属性如下。

- **java.naming.factory.initial**（或 Context.INITIAL_CONTEXT_FACTORY），环境属性名，用于指定 InitialContext 工厂。该属性值应该是创建 InitialContext 的工厂类的全限定类名。如果开发者没有指定其值，则在创建 InitialContext 对象时将会抛出 javax.naming.NoInitialContextException 异常。
- **java.naming.provider.url**（或 Context.PROVIDER_URL），环境属性名，用于指

定客户将使用 JBoss JNDI 服务供应商的位置。NamingContextFactory 类利用该位置信息可以知道其将连接到的 JBossNS 服务器。其值为 URL 字符串，格式为“jnp://host:port/[jndi_path]”。其中，该 URL 的“jnp:”部分是指使用的协议，即 JBoss 使用的基于 Socket/RMI 的协议。URL 的“jndi_path”部分是相对于根上下文的可选 JNDI 名。比如，“apps”或“apps/tmp”。除 host 之外，其他部分都是可选的。由于默认 port 值为 1099，因此如下给出的实例都是等效的。

- jnp://www.jboss.org:1099/
- www.jboss.org:1099
- www.jboss.org
- **java.naming.factory.url.pkgs**（或 Context.URL_PKG_PREFIXES）：环境属性名，在 URL 上下文工厂中装入包前缀列表时，需要使用该属性名。该属性值应该使用“:”隔开的前缀列表。其中，该列表描述了创建 URL 上下文工厂的工厂类的包前缀。对于 JBoss JNDI 供应商而言，其值必须是 org.jboss.naming:org.jnp.interfaces。为定位 JBoss JNDI 供应商的 jnp: 和 java: URL 上下文工厂，开发者必须使用该属性。
- **jnp.socketFactory**：javax.net.SocketFactory 实现的全限定类名，用于创建引导 Socket。其默认值是，org.jnp.interfaces.TimedSocketFactory。TimedSocketFactory 是 SocketFactory 的简单实现，即只支持连接和读超时。它们支持的两个属性如下。
 - jnp.timeout: 以毫秒表示连接超时。默认值为 0，即表示连接阻塞到 JVM TCP/IP 层超时为止。
 - jnp.sotimeout: 以毫秒表示已连接 Socket 的读操作超时。默认值为 0，即表示读操作可能会阻塞。其中，传给新连接 Socket 的 Socket.setSoTimeout 方法的参数就是 jnp.sotimeout 取值。

当客户应用利用上述可用 JBossNS 属性创建 InitialContext 时，将使用对象 org.jnp.interfaces.NamingContextFactory 创建后续操作将使用到的 Context 实例。NamingContextFactory 是 javax.naming.spi.InitialContextFactory 接口的 JBossNS 实现。当告知 NamingContextFactory 创建 Context 时，它将使用 InitialContext 环境和全局 JNDI 命名空间中的上下文名创建 org.jnp.interfaces.NamingContext 实例。其中，实现了 Context 接口的 NamingContext 实例是实际完成连接到 JBossNS 服务器任务的对象。来自环境属性的 Context.PROVIDER_URL 信息表明了服务器是从哪里获得 NamingServer RMI 引用的。

当 NamingContext 实例初次完成了 Context 操作时，它关联 NamingServer 实例实际上是一种惰性行为。当完成 Context 操作时，它并没有和 NamingServer 关联，而是判断是否定义了如下环境属性，即 Context.PROVIDER_URL。Context.PROVIDER_URL 定义了 Context 使用的 JBossNS 服务器的主机和端口。如果提供了该 URL，则 NamingContext 通过检查自身的静态 map 成员以判断 host 和 port 值对标识的 Naming 实例是否已创建。如果已创建，则简单地使用现有的 Naming 实例。如果没有，则根据 host 和 port 值对创建 Naming 实例。进而，NamingContext 能够使用 java.net.Socket 连接到相应的主机和端口，并从通过读取该 Socket 的 java.rmi.MarshalledObject 获得服务器的 Naming RMI 存根。最后，调用 get 方法。其中，在 NamingContext 服务器的 map 成员中，缓存了上述获得的 Naming 实例，而具体位

置是基于 host 和 port 值对计算的。如果开发者没有提供 Context.PROVIDER_URL，则 NamingContext 只是简单的使用 Main MBean 在 JVM 中设置的 Naming 实例。

Context 接口的 NamingContext 实现将其所要完成的所有操作都委派给与其关联的 Naming 实例。实现了 Naming 接口的 NamingServer 类使用 java.util.Hashtable 存储该 Context。对于特定 JBossNS 服务器的各个不同 JNDI 名而言，都存在惟一的 NamingServer 实例。在任何时刻，都可能有 0 个或多个短暂的 NamingContext 实例引用 NamingServer 实例。NamingContext 的重要目的是作为 Naming 接口适配器的 Context。其中，该适配器负责管理传递给 NamingContext 的 JNDI 名的转换。由于 JNDI 名可能是 URL，或某相对名，因此需要将其转换为其引用的 JBossNS 服务器上下文中的绝对名。NamingContext 的重要功能就是完成这种转换。

1. 群集环境中的命名查找

当 JBoss 运行在群集环境中时，开发者可以不指定 Context.PROVIDER_URL 值，并查找网络以获得可用的命名服务。当然，这只是对运行 all 配置的 JBoss 服务器有效，或等效配置，即部署了 org.jboss.ha.framework.server.ClusterPartition 和 org.jboss.ha.jndi.HANamingService 服务的 JBoss 配置。查找过程如下：发送多播请求包给目的地址/端口，并等待响应码。其中，该响应是 Naming 接口的 HA-RMI 版本。如下的 InitialContext 属性将影响查找配置。

- **jnp.partitionName**: 群集划分名，即查找范围仅限于其指定的群集划分名。如果 JBoss 运行环境存在多个群集，则客户可能会限定对某个特定群集的命名查找。如果没有给出默认值，则表示其接受任何群集响应。
- **jnp.discoveryGroup**: 查找操作发送的多播 IP 地址。其默认值为 230.0.0.4。
- **jnp.discoveryPort**: 查找操作发送的端口。其默认值为 1102。
- **jnp.discoveryTimeout**: 以毫秒计算查找操作等待的响应时间，其默认值为 5000 毫秒，即 5 秒。
- **jnp.disableDiscovery**: 标志位，表明是否触发查找过程。当开发者没有指定 Context.PROVIDER_URL 时，或指定的 URL 中没有定位到有效的命名服务时，会触发查找过程。如果其值为 true，则不会触发查找过程。

2. HTTP InitialContext 工厂实现

JBoss 支持基于 HTTP 方式，而实现对 JNDI 命名服务的访问。从客户使用 JNDI 的角度考虑，这种透明的变化使得客户不用理会它，即客户可以继续使用 JNDI Context 接口。其中，通过 Context 接口的请求操作被转换成 HTTP post，并发送给 Servlet，最终使用 JMX invoke 操作将该请求发送给 NamingService。使用 HTTP 的优势在于，能够更好地处理防火墙和代理设置。另外，还能够基于标准 Servlet 角色，以保护对 JNDI 服务的访问。

为实现基于 HTTP 方式访问 JNDI 服务，开发者需要指定如下工厂实现，即 org.jboss.naming.HttpNamingContextFactory。支持该工厂的 InitialContext 环境属性有。

- **java.naming.factory.initial**（或 Context.INITIAL_CONTEXT_FACTORY），该属性是用于指定 InitialContext 工厂的环境属性。java.naming.factory.initial 取值必须是 org.jboss.naming.HttpNamingContextFactory。

- **java.naming.provider.url** (或 `Context.PROVIDER_URL`), 用于指定 JMX Invoker Servlet 的 HTTP URL。其取值取决于 `http-invoker.sar` 的配置及它包含的 WAR 应用。在默认情况下, 将 JMX Invoker Servlet 放置在 `/invoker/JMXInvokerServlet`。HTTP URL 的完整格式应该以如下形式表示: JBoss Servlet 容器的公开 URL + “`/invoker/JMXInvokerServlet`”。比如:

- `http://www.jboss.org:8080/invoker/JMXInvokerServlet`
- `http://www.jboss.org/invoker/JMXInvokerServlet`
- `https://www.jboss.org/invoker/JMXInvokerServlet`

其中, 第一个实例使用 8080 端口作为 Servlet 访问入口, 第二个实例使用 80 端口, 第三个实例使用加密 SSL 连接到位于 443 端口的标准 HTTPS。

- **java.naming.factory.url.pkgs** (或 `Context.URL_PKG_PREFIXES`), 对于所有 JBoss JNDI 供应商, 该属性取值必须是 `org.jboss.naming:org.jnp.interfaces`。

`HttpNamingContextFactory` 返回的 JNDI Context 实现是这样一种代理, 即将调用请求委派给桥接 Servlet, 再由该 Servlet 通过 JMX 总线将调用请求发送给 `NamingService`, 最后通过 HTTP 将返回的响应进行压包。该代理需要获知其操作桥接 Servlet 的 URL。如果 JBoss Web 服务器提供了众所周知的公开接口, 则该 URL 应该被绑定在服务器端。如果 JBoss Web 服务器位于一个或多个防火墙(或代理)后面, 则该代理不能获知所要求的 URL。对于这种情形, 必须在客户端 JVM 中将该代理设置在系统属性值里面。更多信息, 请开发者参考“3.4.2 基于 HTTP 访问 JNDI”小节的内容。

3. Login InitialContext 工厂实现

由于 JAAS 为登录信息提供了更灵活的框架, 因此 JBoss 一直都没有借助于 `InitialContext` 工厂实现获得登录信息。同时, 这也是为简化与其他应用服务器环境的移植性的一种机制。自从 JBoss-3.0.3 开始, JBoss 添加了一个新的 `InitialContext` 工厂实现, 以获得登录信息。该实现仍然使用了 JAAS, 但在客户端应用中并没有明显地使用 JAAS 接口。

提供这种能力的工厂类为 `org.jboss.security.jndi.LoginInitialContextFactory`。支持该工厂的 `InitialContext` 环境属性如下。

- **java.naming.factory.initial** (或 `Context.INITIAL_CONTEXT_FACTORY`), 环境属性名, 用于指定 `InitialContext` 工厂。其值必须是: `org.jboss.security.jndi.LoginInitialContextFactory`。
- **java.naming.provider.url** (或 `Context.PROVIDER_URL`), 该环境属性设置 `NamingContextFactory` 供应商 URL。其中, `LoginInitialContext` 只是包裹了 `NamingContextFactory`, 即将 JAAS 登录操作添加到现有的 `NamingContextFactory` 中。
- **java.naming.factory.url.pkgs** (或 `Context.URL_PKG_PREFIXES`), 对于所有 JBoss JNDI 供应商而言, 其值必须为 `org.jboss.naming:org.jnp.interfaces`。在定位 JBoss JNDI 供应商 `jnp:` 和 `java:` 上下文工厂时, 需要使用到该属性。
- **java.naming.security.principal** (或 `Context.SECURITY_PRINCIPAL`), 认证

Principal。该属性值或者是 `java.security.Principal` 实现，或者是用字符串形式表示的 Principal 名。

- **java.naming.security.credentials**（或 `Context.SECURITY_CREDENTIALS`），用于认证 Principal 的凭证，比如口令、会话密钥等。
- **java.naming.security.protocol**（或 `Context.SECURITY_PROTOCOL`），JAAS 登录模块名，用于 Principal 和凭证的认证。

3.2.2 基于 HTTP 访问 JNDI

除了使用遗留 Socket 引导协议的 RMI/JRMP 外，JBoss-3.0.3 还为基于 HTTP 方式访问 JNDI 命名服务提供支持。所有的这些功能都是由 `http-invoker.sar` 部署应用及其包含的服务和 Servlet 提供的。`http-invoker.sar` 的结构如下。

```
http-invoker.sar
+- META-INF/jboss-service.xml
+- invoker.war
| +- WEB-INF/jboss-web.xml
| +- WEB-INF/classes/org/jboss/invoke/http/servlet/InvokerServlet.class
| +- WEB-INF/classes/org/jboss/invoke/http/servlet/NamingFactoryServlet.class
| +- WEB-INF/classes/org/jboss/invoke/http/servlet/ReadOnlyAccessFilter.class
| +- WEB-INF/classes/roles.properties
| +- WEB-INF/classes/users.properties
| +- WEB-INF/web.xml
| +- META-INF/MANIFEST.MF
+- META-INF/MANIFEST.MF
```

`http-invoker.sar` 中的 `jboss-service.xml` 描述符定义了 `HttpInvoker` 和 `HttpInvokerHAMBean`。这些服务处理借助于 HTTP 将方法调用发送给位于 JMX 总线上的合适目标 MBean 的路由操作。

`http-invoker.war` Web 应用含有的 Servlet 能够处理 HTTP 传输细节。`NamingFactoryServlet` 处理 JBoss JNDI 命名服务中的 `javax.naming.Context` 实现请求。`InvokerServlet` 处理 RMI/HTTP 客户触发的调用。`ReadOnlyAccessFilter` 实现了 JNDI 命名服务的安全性访问，即为未认证客户提供单一的、只读的 JNDI 上下文。

在浏览 `http-invoker` 的配置信息之前，本文先阐述 `http-invoker` 服务的大体操作。图 3-6 给出了逻辑视图描述，即 JBoss JNDI 代理的结构及其与 `http-invoker` 的服务器端组件的关系。将 `InitialContext` 中的 `Context.INITIAL_CONTEXT_FACTORY` 属性设为 `org.jboss.naming.HttpNamingContextFactory`，`Context.PROVIDER_URL` 属性设为 `NamingFactoryServlet` 的 HTTP URL，然后通过 `NamingFactoryServlet` 能够获得该 JNDI 代理。返回的代理嵌入到 `org.jnp.interfaces.NamingContext` 实例中，它实现了 `Context` 接口。

该代理是 `org.jboss.invoke.http.interfaces.HttpInvokerProxy` 实例，而且它还实现了 `org.jnp.interfaces.Naming` 接口。`HttpInvokerProxy` 内部含有的 `Invoker` 将 `Naming` 接口方法调用压包，并借助于 HTTP post 发送给 `InvokerServlet`。`InvokerServlet` 将这些 post 转换成 JMX

Invocation, 即发送给 NamingService。最终, 将响应结果返回给代理, 它位于 HTTP post 的 response 中。

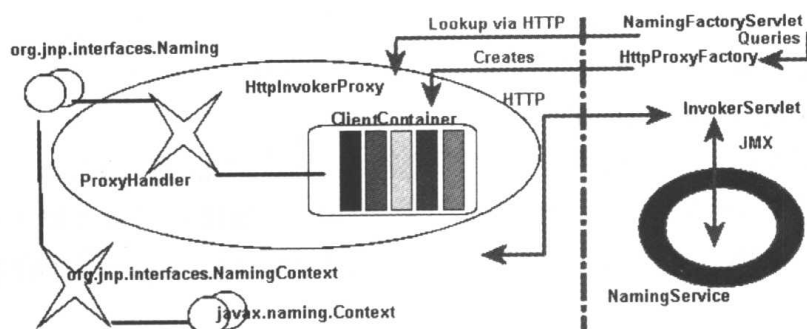


图 3-6 用于 JNDI 上下文的 Http-Invoker 代理/服务器结构

为将这些组件捆在一起, 开发者需要设置若干个配置值。图 3-7 给出了这些配置文件和相应组件的相互关系。

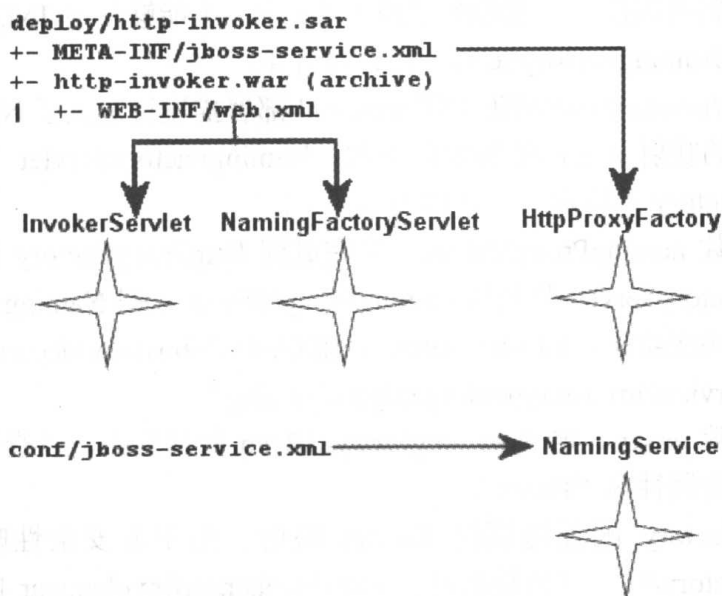


图 3-7 配置文件和 JNDI/HTTP 组件之间的相互关系

http-invoker.sar/META-INF/jboss-service.xml 描述符定义了 HttpProxyFactory, 即为 NamingService 创建 HttpInvokerProxy。需要为 HttpProxyFactory 配置的属性如下。

- **InvokerName**, 定义在 conf/jboss-service.xml 描述符中 NamingService 的 JMX ObjectName。JBoss 发布版使用的标准设置为 “jboss:service=Naming”。
- **InvokerURL**, 或 InvokerURLPrefix + UseHostName + InvokerURLSuffix。可以使用 InvokerURL 指定 InvokerServlet 的完整 HTTP URL, 或者指定主机名、各个 URL 组件, 并将 HttpProxyFactory 填充到其中。比如, InvokerURL 实例, “http://jboss-host1.dot.com:8080/invoker/JMXInvokerServlet”。分隔其各个组件。
 - **InvokerURLPrefix**: hostname 前的 url 前缀。一般是, “http://”, 或使用 SSL 的 “https://”。比如, 上述 InvokerURL 实例值的 InvokerURLPrefix 是 “http://” (不包括双引号)。

- **InvokerURLSuffix**: hostname 后的 url 后缀。其包括 Web 服务器的端口号及 InvokerServlet 的部署路径。比如, 上述 InvokerURL 实例值的 InvokerURLSuffix 为 “:8080/invoker/JMXInvokerServlet” (不包括双引号)。其中, 端口号的设置由 Web 容器服务决定。InvokerServlet 的部署路径由 http-invoker.sar/invoker.war/WEB-INF/web.xml 描述符设置。
- **UseHostName**: 标志位, 表明完整 InvokerURL 的 hostname 部分是否使用主机名, 还是主机 IP 地址。如果为 true, 则使用 InetAddress.getLocalHost().getHostName 方法; 否则, 使用 InetAddress.getLocalHost().getHostAddress() 方法。
- **ExportedInterface**: 代理暴露给客户的 org.jnp.interfaces.Naming 接口。其中, JBoss JNDI 实现, 即 NamingContext 类是该代理的实际客户。当使用 JBoss JNDI 供应商查找命名服务时, JNDI 客户使用 InitialContext 获得 NamingContext 类。
- **JndiName**: JndiName 属性指定, 代理绑定到的 JNDI 名。需要将其值设为空字符串, 即表明该接口不应该绑定到 JNDI 中。不能够使用 JNDI 引导该代理本身, 而是通过 NamingFactoryServlet 完成其引导。

http-invoker.sar/invoker.war/WEB-INF/web.xml 部署描述符定义了 NamingFactoryServlet 和 InvokerServlet 的映射及它们的初始化参数。NamingFactoryServlet 中有关 JNDI/HTTP 的配置位于 JNDIFactory 入口项中, 具体内容如下。

- 初始化参数 namingProxyMBean。它完成到 HttpProxyFactory MBean 名的映射。NamingFactoryServlet 将使用 namingProxyMBean 获得 Naming 代理, 即 HTTP post 所返回的响应结果。http-invoker.sar/META-INF/jboss-service.xml 对应的默认值为 “jboss:service=invoke,type=http,target=Naming”。
- 初始化参数 proxy。定义 namingProxyMBean 的属性名, 以查询 Naming 代理值。其默认值为属性名 “Proxy”。
- 为 JNDIFactory 配置提供的 Servlet 映射。用于非安全性映射的默认设置是 “JNDIFactory/*”。该值是相对于 http-invoker.sar/invoker.war 的根上下文。其中, 这里的 war 名字指 “.war” 后缀。

InvokerServlet 中有关 JNDI/HTTP 的配置位于 JMXInvokerServlet 入口项中, 具体如下:

- InvokerServlet 的 Servlet 映射。用于非安全性映射的默认设置是 “InvokerServlet/*”。该值是相对于 http-invoker.sar/invoker.war 的根上下文。其中, 这里的 war 名字指 “.war” 后缀。

1. 基于 HTTPS 访问 JNDI

为实现在 HTTP/SSL 基础上访问 JNDI, 需要配置好 Web 容器的 SSL 连接器。在 “第 9 章 集成 Servlet 容器” 中, 本书将会介绍在 Tomcat 中配置 SSL 连接器。在此将给出一个简单的客户应用实例, 即使用 HTTPS URL 作为 JNDI 供应商 URL, 以演示 HTTPS 的使用。另外, 如果开发者对 SSL 连接器设置的详细配置感兴趣, 本文还给出了 SSL 连接器配置, 并且它还是一完整实例。

同时, 本节也讲解了 HttpProxyFactory 的配置, 以使用 HTTPS URL, 如列表 3-16 所

示。其中,本实例也使用了 http-invoker.sar 中的 jboss-service.xml 描述符。所有与标准 HTTPS 配置的不同之处在于 InvokerURLPrefix 和 InvokerURLSuffix 的属性,它们使用了位于 8443 端口的 HTTPS URL。

列表 3-16 HttpPoxeyFactory 配置实例

```
<!-- Expose the Naming service interface via HTTPS -->
<mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
  name="jboss:service=invoker,type=https,target=Naming">
  <!-- The Naming service we are proxying -->
  <attribute name="InvokerName">jboss:service=Naming</attribute>
  <!-- Compose the invoker URL from the cluster node address -->
  <attribute name="InvokerURLPrefix">https://</attribute>
  <attribute name="InvokerURLSuffix">:8443/invoker/JMXInvokerServlet
</attribute>
  <attribute name="UseHostName">true</attribute>
  <attribute name="ExportedInterface">org.jnp.interfaces.Naming
</attribute>
  <attribute name="JndiName"></attribute>
  <attribute name="ClientInterceptors">
    <interceptors>
      <interceptor>org.jboss.proxy.ClientMethodInterceptor
    </interceptor>
      <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
      <interceptor>org.jboss.naming.interceptors.ExceptionInterceptor
    </interceptor>
      <interceptor>org.jboss.invocation.InvokerInterceptor
    </interceptor>
    </interceptors>
  </attribute>
</mbean>
```

为使用 HTTPS, JNDI 客户应用要求, 至少要设置 HTTPS URL 协议处理器。本节使用 JSSE 实现 HTTPS。通过查看 JSSE 文档获悉, 配置实例客户应用 (如列表 3-17 所示) 的步骤如下:

步骤

(1) 必须提供 Java 可用的、用于 HTTPS URL 的协议处理器。JSSE 发布版在其提供的 com.sun.net.ssl.internal.www.protocol 包中提供了 HTTPS 处理器。为了能够使用 HTTPS URL, 需要将该包包含在标准 URL 协议处理器的搜索属性 java.protocol.handler.pkgs 中。这里将该属性设置在 Ant 脚本中。

(2) 为生效 SSL, 开发者需要安装 JSSE 安全性供应商。可以将 JSSE jar 包安装为扩展包, 或者程序实现。为了更能说明问题, 本实例通过程序来实现。ExClient 代码的第 18 行给出了相应的实现。

(3) JNDI 供应商 URL 必须使用协议 HTTPS。ExClient 代码的第 24~25 行指定了到主机为 localhost、端口为 8443 的 HTTP/SSL 连接。Web 容器的 SSL 连接器定义了主机和端口。

(4) 将基于服务器证书的 HTTPS URL 主机验证失效。在默认情况下，JSSE HTTPS 协议处理器基于服务器证书的公共名对 HTTPS URL 的主机部分进行严格验证。当借助于 Web 浏览器连接到加密网站时，浏览器也使用这种方式进行安全性验证。本文使用第 8 章“SSL 实例”中的自签名服务器证书，而不是特定主机名，这也是在应用开发环境或企业内网常见的使用方式。如果检查到系统属性 org.jboss.security.ignoreHttpsHost，并且其值为 true，则 HttpInvokerProxy 将覆盖默认的主机名。本文在 Ant 脚本中将该属性设置为 true。

列表 3-17 使用 HTTPS 传输的 JNDI 客户应用

```
1 package org.jboss.chap3.ex1;
2
3 import java.security.Security;
4 import java.util.Properties;
5 import javax.naming.Context;
6 import javax.naming.InitialContext;
7
8 /** A simple JNDI client that uses HTTPS as the transport.
9  *
10  * @author Scott.Stark@jboss.org
11  * @version $Revision: 1.1 $
12  */
13 public class ExClient
14 {
15     public static void main(String args[]) throws Exception
16     {
17         // Install the Sun JSSE provider since we may not have JSSE installed
18         Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
19         System.out.println("Added JSSE security provider");
20
21         Properties env = new Properties();
22         env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
23             "org.jboss.naming.HttpNamingContextFactory");
24         env.setProperty(Context.PROVIDER_URL,
25             "https://localhost:8443/invoke/JNDIFactory");
26         Context ctx = new InitialContext(env);
27         System.out.println("Created InitialContext, env="+env);
28         Object data = ctx.lookup("jmx/rmi/RMIAdaptor");
29         System.out.println("lookup(jmx/rmi/RMIAdaptor): "+data);
30     }
31 }
```

为测试列表 3-17 给出的客户应用，首先创建 chap3 配置文件集合。


```
[nr@toki examples]$ ant -Dchap=chap3 config
Buildfile: build.xml

validate:
    [java] ImplementationTitle: JBoss [WonderLand]
    [java] ImplementationVendor: JBoss.org
    [java] ImplementationVersion: 3.2.3 (build: CVSTag=JBoss_3_2_3 date=200311301445)
    [java] SpecificationTitle: JBoss
    [java] SpecificationVendor: JBoss (http://www.jboss.org/)
    [java] SpecificationVersion: 3.2.3
    [java] JBoss version is: 3.2.3

fail_if_not_valid:

init:
    [echo] Using jboss.dist=/tmp/jboss-3.2.3

compile:

config:

config:
    [echo] Preparing chap3 configuration fileset
    [mkdir] Created dir: /tmp/jboss-3.2.3/server/chap3
    [copy] Copying 151 files to /tmp/jboss-3.2.3/server/chap3
    [copy] Copied 3 empty directories to /tmp/jboss-3.2.3/server/chap3
    [copy] Copying 1 file to /tmp/jboss-3.2.3/server/chap3/conf
    [copy] Copying 1 file to /tmp/jboss-3.2.3/server/chap3/conf
    [copy] Copying 1 file to /tmp/jboss-3.2.3/server/chap3/deploy/jbossweb-tomcat41.sar/META-INF
    [copy] Copying 1 file to /tmp/jboss-3.2.3/server/chap3/deploy/http-invoker.sar/META-INF
    [copy] Copying 1 file to /tmp/jboss-3.2.3/server/chap3/deploy/http-invoker.sar/invoker.war/
    WEB-INF

BUILD SUCCESSFUL
Total time: 6 seconds
```

接下来，使用 chap3 配置文件集合启动 JBoss 服务器。

```
[nr@toki bin] ./run.sh -c chap3
=====

JBoss Bootstrap Environment
....
```

最后，运行 ExClient 客户端。

```
[nr@toki examples]$ ant -Dchap=chap3 -Dex=1 run-example
Buildfile: build.xml
```



```
validate:
    [java] ImplementationTitle: JBoss [WonderLand]
    [java] ImplementationVendor: JBoss.org
    [java] ImplementationVersion: 3.2.3 (build: CVSTag=JBoss_3_2_3 date=200311301445)
    [java] SpecificationTitle: JBoss
    [java] SpecificationVendor: JBoss (http://www.jboss.org/)
    [java] SpecificationVersion: 3.2.3
    [java] JBoss version is: 3.2.3

fail_if_not_valid:

init:
    [echo] Using jboss.dist=/tmp/jboss-3.2.3

compile:

run-example:

run-example1:
    [java] JSSE already available
    [java] Created InitialContext, env={java.naming.provider.url=https://localhost:8443/invoker/
    JNDIFactorySSL, java.naming.factory.initial=org.jboss.naming.HttpNamingContextFactory}
    [java] lookup(jmx/rmi/RMIAdaptor): org.jboss.invocation.jmp.interfaces.
    JRMPInvokerProxy@781288

BUILD SUCCESSFUL
Total time: 6 seconds
```

3.2.3 保护基于 HTTP 访问 JNDI

基于 HTTP 访问 JNDI 的优势之一在于除了很容易保护对 JNDI InitialContext 工厂的访问外，还能够使用标准的 Web 安全性声明实现命名操作。由于 JNDI/HTTP 传输的服务端处理是由两个 Servlet 实现的，因此上述实现还是有可能的。同时，开发者通过 default 和 all 配置的 deploy 目录能够找到这两个 Servlet，即位于 http-invoker.sar/invoker.war 目录。为生效对 JNDI 访问的保护操作，开发者需要编辑描述符 invoker.war/WEB-INF/web.xml，并删除所有的未保护 Servlet 映射。比如，列表 3-18 给出了 web.xml 描述符，如果用户能够通过认证，并且其角色为 HttpInvoker，则允许该用户访问 invoker.war 中的 Servlet。

列表 3-18 用于保护访问 JNDI Servlet 的 web.xml 描述符实例

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
```

```

<web-app>
<!-- ### Servlets -->
  <servlet>
    <servlet-name>JMXInvokerServlet</servlet-name>
    <servlet-class>org.jboss.invocation.http.servlet.InvokerServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet>
    <servlet-name>JNDIFactory</servlet-name>
    <servlet-class>org.jboss.invocation.http.servlet.NamingFactoryServlet</servlet-class>
  <init-param>
    <param-name>namingProxyMBean</param-name>
    <param-value>jboss:service=invoker,type=http,target=Naming</param-value>
  </init-param>
  <init-param>
    <param-name>proxyAttribute</param-name>
    <param-value>Proxy</param-value>
  </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>

<!-- ### Servlet Mappings -->
  <servlet-mapping>
    <servlet-name>JNDIFactory</servlet-name>
    <url-pattern>/restricted/JNDIFactory/*</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>JMXInvokerServlet</servlet-name>
    <url-pattern>/restricted/JMXInvokerServlet/*</url-pattern>
  </servlet-mapping>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>HttpInvokers</web-resource-name>
      <description>An example security config that only allows users with the role
      HttpInvoker to access the HTTP invoker servlets
      </description>
      <url-pattern>/restricted/*</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>HttpInvoker</role-name>
    </auth-constraint>
  </security-constraint>

```

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>JBoss HTTP Invoker</realm-name>
</login-config>

<security-role>
  <role-name>HttpInvoker</role-name>
</security-role>
</web-app>
```

web.xml 描述符仅仅定义了保护的 Servlet，以及能够访问受保护 Servlet 的角色名。另外，开发者还需要为 WAR 定义安全域，才能够处理认证和授权。通过描述符 jboss-web.xml 能够达到此目的。如下 jboss-web.xml 描述符实例给出了“http-invoker”安全域。

```
<jboss-web>
  <security-domain>java:/jaas/http-invoker</security-domain>
</jboss-web>
```

security-domain 元素定义了安全域的名字，即用于 JAAS 登录模块的认证和授权操作。在“8.1.6 使用 JBoss 中的安全性声明”小节中将提供更详细的资料。

3.2.4 保护只读、未保护上下文 JNDI 的访问

JNDI/HTTP 命名服务的另一特性是开发者能够定义上下文，即允许未认证用户以只读模式访问的上下文。这是认证层使用的重要服务。比如，为完成认证工作，模块 SRPLoginModule 需要查找 SRP 服务器接口。因此，开发者需要对 web.xml 描述符新增其他的设置信息。列表 3-19 给出了这些必要的元素。

列表 3-19 只读访问所需的其他 web.xml 描述符元素

```
<web-app>
<filter>
  <filter-name>ReadOnlyAccessFilter</filter-name>
  <filter-class>org.jboss.invocation.http.servlet.ReadOnlyAccessFilter</filter-class>
  <init-param>
    <param-name>readOnlyContext</param-name>
    <param-value>readonly-context</param-value>
  </init-param>
  <init-param>
    <param-name>invokerName</param-name>
    <param-value>jboss:service=Naming</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>ReadOnlyAccessFilter</filter-name>
  <url-pattern>/readonly/*</url-pattern>
```

```

</filter-mapping>

<servlet>
  <servlet-name>ReadOnlyJNDIFactory</servlet-name>
  <servletclass>org.jboss.invocation.http.servlet.NamingFactoryServlet</servlet-class>
  <init-param>
    <param-name>namingProxyMBean</param-name>
    <paramvalue>jboss:service=invoker,type=http,target=Naming,readonly=true</paramvalue>
  </init-param>
  <init-param>
    <param-name>proxyAttribute</param-name>
    <param-value>Proxy</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>

<!-- A mapping for the JMXInvokerServlet that only allows invocations
of lookups under a read-only context. This is enforced by the
ReadOnlyAccessFilter
-->
<servlet-mapping>
  <servlet-name>JMXInvokerServlet</servlet-name>
  <url-pattern>/readonly/JMXInvokerServlet/*</url-pattern>
</servlet-mapping>

```

通过上述设置，客户能够对“readonly-context”或其子上下文进行 Context.lookup 操作，除此之外不能对该上下文进行其他操作。同时，也不能对其他上下文进行任何操作。下面给出的代码片段，讲解了“readonly-context/data”绑定的查找操作。

```

Properties env = new Properties();
env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
    "org.jboss.naming.HttpNamingContextFactory");
env.setProperty(Context.PROVIDER_URL,
    "http://localhost:8080/invoker/ReadOnlyJNDIFactory");
Context ctx2 = new InitialContext(env);
Object data = ctx2.lookup("readonly-context/data");

```

3.2.5 其他命名 MBean

除了 NamingService MBean，即在 JBoss 中配置嵌入式 JBossNS 服务器，随 JBoss 发布的、与命名相关的 MBean 服务还有 3 个。它们分别是：ExternalContext、NamingAlias 及 JNDIView。

1. org.jboss.naming.ExternalContext MBean

ExternalContext MBean，能够集成外部 JNDI 上下文到 JBoss 服务器的 JNDI 命名空间中。其中，这里的“外部”指运行 JBoss 服务器 JVM 内部的 JBossNS 命名服务之外的任

何命名服务。开发者能够集成 LDAP 服务器、文件系统、DNS 服务器等，甚至是那些上下文不支持序列化的 JNDI 供应商。如果命名服务支持远程访问，这种集成性使得各种 JNDI 供应商能供远程客户使用。

为使用外部 JNDI 命名服务，开发者需要将 ExternalContext MBean 配置添加给 jboss.jcml 配置文件。ExternalContext 服务的可配置属性如下。

- **JndiName:** 外部上下文绑定的 JNDI 名。
- **RemoteAccess:** boolean 标志位，表明是否使用 Serializable 形式绑定外部 InitialContext，从而允许远程客户创建外部 InitialContext。当远程客户借助于 JBoss JNDI InitialContext 查找外部上下文时，使用传递给 ExternalContext MBean 的环境属性，能够有效地创建外部 InitialContext 实例。只有客户确实能远程创建新的 “InitialContext(env)”，才能完成上述目的。当然，也要求访问上下文的远程 VM 能够解析环境属性，即 Context.PROVIDER_URL 的值。LDAP 实例能满足上述条件。对于文件系统实例，除非文件系统路径指向网络路径，否则满足不了上述条件。其默认值为 false。
- **CacheContext:** 标志位。如果为 true，则只有在 MBean 启动时才会创建外部 Context，并将其存储成内存对象，直到 MBean 停止。如果为 false，则每次查找时，都会使用 MBean 属性和 InitialContext 类创建 Context。当没有缓存客户查找到的 Context 时，客户应该调用 Context 的 close() 方法，以防止资源泄漏。
- **InitialContext:** 使用 InitialContext 实现的全限定类名。其取值必须是如下之一：javax.naming.InitialContext、javax.naming.directory.InitialDirContext、javax.naming.ldap.InitialLdapContext。如果是 InitialLdapContext，则将使用值为 null 的 Controls 数组。默认值为 javax.naming.InitialContext。
- **Properties:** 为外部 InitialContext 设置 jndi.properties。其取值可以是 URL 字符串，也可以是为 classpath 资源名。实例如下：
 - file:///config/myldap.properties
 - http://config.mycompany.com/myldap.properties
 - /conf/myldap.properties
 - myldap.properties

列表 3-20 给出了两个 jboss.jcml 配置文件片段，一个用于 LDAP 服务器，另一个用于本地文件系统目录。

列表 3-20 ExternalContext MBean 配置

```
<!-- Bind a remote LDAP server -->
<mbean code="org.jboss.naming.ExternalContext"
  name="jboss.jndi:service=ExternalContext,jndiName=external/ldap/jboss">
  <attribute name="JndiName">external/ldap/jboss</attribute>
  <attribute name="Properties">jboss.ldap</attribute>
  <attribute name="InitialContext">
    javax.naming.ldap.InitialLdapContext
  </attribute>
```



```
<attribute name="RemoteAccess">true</attribute>
</mbean>
<!-- Bind the /usr/local file system directory -->
<mbean code="org.jboss.naming.ExternalContext"
  name="jboss.jndi:service=ExternalContext,jndiName=external/fs/usr/local" >
  <attribute name="JndiName">external/fs/usr/local</attribute>
  <attribute name="Properties">local.props</attribute>
  <attribute name="InitialContext">javax.naming.InitialContext</attribute>
</mbean>
```

第一个配置描述了将外部 LDAP 上下文集成到 JBoss JNDI 命名空间“external/ldap/jboss”。实例 jboss.ldap 的属性文件如下。

```
java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
java.naming.provider.url=ldap://ldaphost.jboss.org:389/
o=jboss.org
java.naming.security.principal=cn=Directory Manager
java.naming.security.authentication=simple
java.naming.security.credentials=secret
```

通过该配置，开发者能够使用下列代码片段从 JBoss VM 内部访问位于 ldap://ldaphost.jboss.org:389/o=jboss.org 的外部 LDAP 上下文。

```
InitialContext iniCtx = new InitialContext();
LdapContext ldapCtx = iniCtx.lookup("external/ldap/jboss");
```

由于 RemoteAccess 属性值为 true，因此在 JBoss 服务器 JVM 外部能够运行上述代码片段。如果 RemoteAccess 属性值为 false，则不能运行，因为远程客户接受到的引用对象 ObjectFactory 不能够重新创建外部 InitialContext。

第二个配置描述了将本地文件系统目录/usr/local 绑定到 JBoss JNDI 命名空间中的“external/fs/usr/local”名。实例 local.props 的属性文件如下。

```
java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory
java.naming.provider.url=file:///usr/local
```

开发者通过该配置能够从 JBoss VM 中使用如下代码片段访问到位于 file:///usr/local 的外部文件系统上下文。

```
InitialContext iniCtx = new InitialContext();
Context ldapCtx = iniCtx.lookup("external/fs/usr/local");
```

2. org.jboss.naming.NamingAlias MBean

实用服务，即 NamingAlias MBean 允许开发者以 JNDI javax.naming.LinkRef 形式创建从某 JNDI 名到另一 JNDI 名的别称。这类似于 UNIX 文件系统特征连接（symbolic link）。对于每个别称，开发者需要向 jboss.jcml 配置文件添加 NamingAlias MBean 的配置属性。其中，NamingAlias 服务的可配置属性如下。

- **FromName**，LinkRef 绑定在 JNDI 中的位置信息。

- **ToName**, 别称的 to 名字。LinkRef 引用的目标名字。该名字或者为 URL, 或者为相对于 InitialContext 解析的名字。如果名字的首字母为 “.”, 则也有可能为相对于该连接绑定到的上下文的名字。

如下给出了一个实例, 该实例将 JNDI 名“QueueConnectionFactory”映射成“Connection Factory”。

```
<mbean code="org.jboss.naming.NamingAlias"
  name="jboss.mq:service=NamingAlias,fromName=QueueConnectionFactory">
  <attribute name="ToName">ConnectionFactory</attribute>
  <attribute name="FromName">QueueConnectionFactory</attribute>
</mbean>
```

3. org.jboss.naming.JNDIView MBean

JNDIView MBean, 允许用户使用 JMX 代理视图接口浏览 JBoss 服务器上存在的 JNDI 命名空间树。为使用 JNDIView 服务, 只需要往 jboss.jcml 文件中添加相应的配置。由于 JNDIView 服务没有可配置属性, 因此合适的配置实例如下。

```
<mbean code="org.jboss.naming.JNDIView" name="jboss:service=JNDIView"/>
```

为了能够使用 JNDIView MBean 浏览 JBoss JNDI 命名空间, 开发者需要使用 HTTP 接口连接到 JMX 代理视图。其默认位置为 <http://localhost:8080/jmx-console/index.jsp>。通过该页面, 开发者能看到有一区域根据域划分列举出了注册的 MBean。图 3-8 给出了实例, 即已配置 JBoss MBean 的 HTTP JMX 代理视图, 鼠标指向的是 JNDIView MBean。

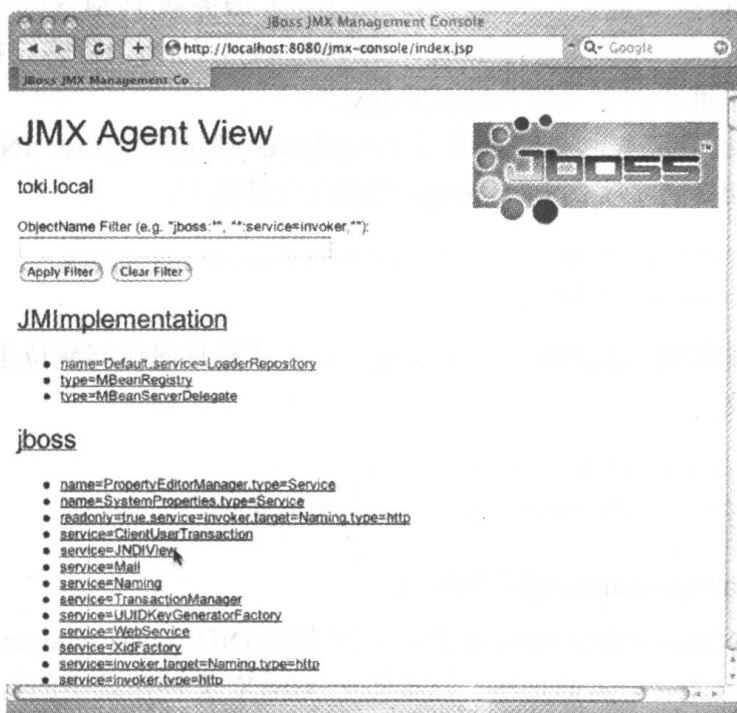


图 3-8 已配置 JBoss MBean 的 HTTP JMX 代理视图

选择 JNDIView 连接, 便能带领用户领略到 JNDIView MBean 视图, 其中列举出了

JNDIView MBean 操作列表。图 3-9 给出了实例, 即 JNDIView MBean 的 HTTP JMX MBean 视图。

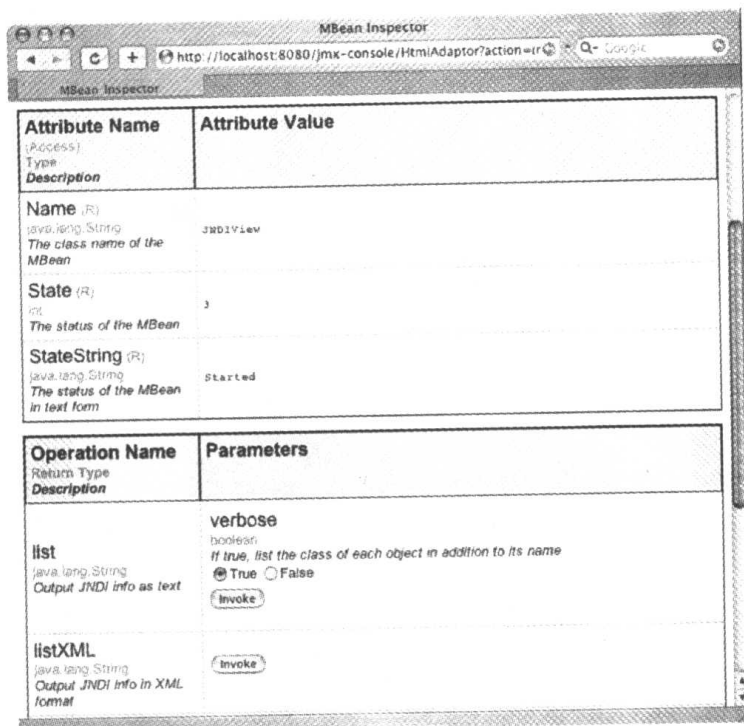


图 3-9 JNDIView MBean 的 HTTP JMX MBean 视图

其中，list 操作将以 HTML 页面形式，即简单的文本视图，打印出 JBoss 服务器 JNDI 命名空间。比如，对于默认 JBoss 3.0.1 发布版服务器而言，list 操作的相应结果如图 3-10 所示。



图 3-10 JNDIView list 操作的 HTTP JMX 输出视图

第4章 JBoss 之事务——JTA 事务服务

本章讨论 JBoss 中的事务管理和 JBossTX 架构。JBossTX 架构允许 JBoss 使用任何 JTA 事务管理器实现。同时，JBossTX 本身也包含了一个兼容 JTA 标准的、快速嵌入虚拟机式的事务管理器实现，以作为默认的事务管理器。首先，本章将介绍一些 JTA 中的重要事务概念和理论，从而为更好地理解 JBossTX 架构提供足够的背景知识。然后，将继续讨论组成 JBossTX 架构的具体接口。最后，本章将讨论 MBean，以用于集成不同的事务管理器。

4.1 事务/JTA 概述

本文将事务定义为包含一个或多个操作的工作单元。其中，这些工作单元涉及到具备 ACID 属性的若干个共享资源。ACID 是原子性、一致性、隔离性及持久性的缩写，它们是事务的四个重要属性。这些术语的具体含义分别如下。

- 原子性：事务必须是原子的。意思是说，或者事务中的所有操作都成功完成，或者都失败。部分成功，或者部分失败是不允许的。
- 一致性：当事务完成时，系统必须处于稳定的、一致的状态。
- 隔离性：不同的事务必须相互隔离。含义为，如果某个事务还未提交但已经完成了一部分操作，则其他事务是看不到的。同时，对于多用户的系统而言，其编程方式就如同单一的进程访问系统一样。
- 持久性：当事务成功提交时，其操作期间引起的变化应该是持久的。如果事务提交，即使服务器瘫痪，这种变化都是不可丢失的。

为阐述这些概念，可以考虑某个简单的银行储蓄应用。如果银行应用程序通过数据库存储了多个账户，而这些账户所存储的货币总和必须总是为 0。当有数量为 M 大小的货币从账户 A 过户到账户 B 时，必须从账户 A 减掉 M，并将账户 B 加上 M。该操作必须在事务中完成，才能够保证 ACID 属性的正确性。

原子性属性意味着取钱和存钱一起构成了不可分割的单元。如果基于某种原因不能完成这两个操作，则结果是对 A、B 两个账户不会造成任何影响。

一致性属性意味着在事务完成后，所有账户的货币总和必须仍然为 0。

当同时有多个银行职员操作某个账户时，隔离性属性显得很重要。取钱或存钱操作可以通过如下三个步骤实现。首先，从数据库读出账户的存款数。其次，对读出的存款数进行加或减。最后，新的数量写入到数据库中。如果没有事务隔离性，会发生很多古怪的结果。比如，如果两个进程同时读取账户 A 的存款数量，并且它们在将新的数量写入数据库之前各自独立地进行了加减操作，那么第一个变更将会错误地被第二个变更覆盖。

当然，持久性也是很重要的。如果货币过户事务提交了，银行必须保证随后的失败并不能取消上次过户的货币。

4.1.1 悲观锁/乐观锁

在通常情况下，事务隔离都是通过锁住被事务访问的任何共享资源。但是，存在两种截然不同的方式来处理事务锁：悲观锁和乐观锁。

悲观锁的优势在于如果事务没有提交，那么它锁住的资源同时也不能够被其他的事务访问，直到事务的完成。如果大部分事务只是简单的读取资源而不更改它们，则这种排外锁会造成性能的下降，并引起锁的争夺。因此，使用乐观锁可能是更好的办法。对于悲观锁而言，锁处于自动纠错状态。比如，上述的银行例子中，一旦有事务访问某个账户，则该账户立即被锁住。其他试图访问该账户的事务或者因为等待该账户的锁被释放而耽误时间，或者本身回滚。只有事务提交或回滚后，锁才会被释放。

而对于乐观锁，当某个资源初次被某事务访问时，它实际上并没有被锁住。结果恰好相反，即当乐观锁锁住资源的同时，该资源的状态也被保存下来。其他的事务能够并发访问该资源，因此引起冲突的变更很可能发生。在提交的过程中，如果需要更新持久源中的资源，则该资源的状态信息将被再次读出，并和该资源被初次访问时的状态进行比较，如果两个状态不一样，则会激发冲突，从而引起事务回滚。

在上述银行应用的例子中，当某事务初次访问某账户时，该账户的存款数量被保存下来。一旦事务即将变更该账户时，其先前保存的存款数量被再次读取出来。如果在事务进行中，该数量发生了变更，事务将失败。否则，新的数量将写入到持久源中。

4.1.2 分布式事务的组件

分布式事务的参与者有很多，具体如下。

- 事务管理器：该组件分布在事务性系统中。它主要是管理和调整涉及到事务的工作。同时，事务管理器以 JTA 中的 `javax.transaction.TransactionManager` 接口形式存在。
- 事务上下文：它表示某特定的事务。在 JTA 中，对应的接口为 `javax.transaction.Transaction`。
- 事务性客户：它能够在某事务中调用一个或多个事务性对象。触发事务的事务性客户称为事务发起者。事务性客户或者显式地使用 JTA 接口，或者隐式地使用。在 JTA 中，没有接口用于表示事务性客户。
- 事务性对象：它是这样一种对象，即在事务性上下文中其行为受到事务操作的影响。事务性对象本身也可以是事务性客户，大部分企业 Bean 都是事务性对象。
- 可恢复资源：它是这样一种对象，即如果事务成功提交，则其状态保存到持久源中；如果事务回滚，则其状态恢复到起初的情形。在事务提交阶段，事务管理器通过使用两阶段 XA 协议和可恢复资源进行通信，以确保多个可恢复资源的事务集成性。事务性数据库和消息中间件，比如 JBossMQ，就是可恢复资源。在 JTA 中，可恢复资源通过接口 `javax.transaction.xa.XAResource` 表示。

4.1.3 两阶段 XA 协议

当事务即将提交时，事务管理器需要保证所有的操作提交，或者全部回滚。如果事务中只涉及到单一的可恢复资源，事务管理器的任务很简单，即告知资源将变化提交到持久源中。

一旦事务涉及到多个可恢复资源时，提交的管理变得更加复杂。简单的告知每个可恢复资源提交各自的变化是不足以维护事务的原子属性的。道理很简单，如果某个可恢复资源提交了，而另一个失败，则事务的部分操作提交了，其他部分就会失败。

通过两阶段 XA 协议可以解决这样的问题。XA 协议在事务的实际提交之前引入了附加的阶段。在告知各可恢复资源提交之前，事务管理器命令所有的可恢复资源准备提交。一旦某可恢复资源表示已经为提交事务做好准备，则表明它能够提交事务。同时，如果需要，资源仍然可以回滚该事务。

因此，组成事务管理器的第一阶段应该是告知所有的可恢复资源准备提交。如果任何可恢复资源失败，则事务将回滚。但是，如果所有的可恢复资源表明能够准备提交，那么 XA 协议的第二阶段就会开始。该阶段告知所有的可恢复资源提交事务，由于所有的可恢复资源都表明已做好准备，因此该步骤不允许失败。

4.1.4 启发式异常

分布式环境中经常会发生通信失败。如果事务管理器和可恢复资源不能够在给定的时间内取得通信，那么可恢复资源可能会单方面提交或回滚该事务上下文中所做的变更。这种情形，称为启发式决定。这是发生于分布式环境中最坏的一种错误，因为它将导致部分事务提交，部分回滚，从而侵犯了事务的原子性，进而引起数据的不一致而导致系统瘫痪。

因为启发式异常存在的这些危险，触发启发式决定的可恢复资源必须将这些决定信息维护在持久源中，直到事务管理器告知它已经忘记这些启发式决定。另外，存储在持久源中有关启发式决定的实际数据依赖于具体的可恢复资源类型，这并没有标准的处理方式。系统管理员或许能够查看这些数据，并且可能编辑资源以纠正任何数据集成问题。

JTA 定义了几种不同的启发式异常。当可恢复资源被告知回滚时，抛出 `javax.transaction.HeuristicCommitException`，以报告启发式决定已经完成。同时，所有相关的更新都已经提交。相反，`javax.transaction.HeuristicRollbackException` 异常的抛出则表明启发式决定已经完成。同时，所有相关的更新都已经回滚。

`javax.transaction.HeuristicMixedException` 是最坏的启发式异常。如果抛出这种异常，则表明事务的部分提交，部分失败。事务管理器抛出这种异常表明一些可恢复资源做了启发式提交，而另一些做了启发式回滚。

4.1.5 事务 ID 和分支

在 JTA 中，事务标识被封装在实现 `javax.transaction.xa.Xid` 接口的对象中。事务 ID 表

示如下三部分的集合。

- 格式标识部分表明该事务家族及其他两部分应该如何解释。
- 全局事务 ID 标识了事务家族中的全局事务。
- 分支标识表明全局事务的特定分支。

事务分支用于标识同一全局事务的部分。无论何时，只要在某一事务中事务管理器涉及到新的可恢复资源，事务管理器将创建新的事务分支。

4.2 JBoss 事务内核

JBoss 应用服务器实现了与实际事务管理器的无关性。JBoss 使用 JTA 中的 `javax.transaction.TransactionManager` 接口作为服务器的事务管理器视图。因此，JBoss 可以使用实现了 JTA `TransactionManager` 接口的任何事务管理器。通过众所周知的 JNDI 定位“`java:/TransactionManager`”可以获得事务管理器。对于服务器事务管理器而言，该 JNDI 是全局可用的访问入口。

如果事务上下文需要和 RMI/JRMP 调用一起使用，那么事务管理器为导入和导出事务传播上下文 (TPC)，则它也必须实现两个简单的接口。其中，这两个接口分别是 `org.jboss.tm.TransactionPropagationContextImporter` 和 `org.jboss.tm.TransactionPropagationContextFactory` 接口。

独立于实际使用的事务管理器也意味着 JBoss 并没有限定 TPC 类型的格式。在 JBoss 中，TPC 是 Object 类型。另外，惟一的需求是 TPC 必须实现 `java.io.Serializable` 接口。

当使用 RMI/JRMP 协议实现远程调用时，TPC 作为类 `org.jboss.ejb.plugins.jrmp.client.RemoteMethodInvocation` 的成员，从而实现远程方法调用请求的发送。

4.2.1 为 JBoss 适配事务管理器

事务管理器必须实现 JTA，从而很容易地实现和 JBoss 集成。当 JBoss 中的一切就绪时，事务管理器作为 MBean 而被管理着。同所有其他的 JBoss 服务一样，它也必须实现 `org.jboss.system.ServiceMBean`，以确保正确的生命周期管理。

启动事务管理器服务的主要需求在于绑定 3 个必需接口的实现到 JNDI 上。这些接口和各自的 JNDI 定位如下：

- `javax.transaction.TransactionManager` 接口。应用服务器使用该接口管理事务，即代表那些容器管理事务 (CMT) 的事务对象。另外，它必须绑定在 JNDI 名“`java:/TransactionManager`”上。
- TPC 工厂接口，`org.jboss.tm.TransactionPropagationContextFactory`。当远程方法调用中需要传输事务的时候，JBoss 通过它来创建 TPC。另外，它必须绑定在 JNDI 名“`java:/TransactionPropagationContextImporter`”上。
- TPC 导入接口，`org.jboss.tm.TransactionPropagationContextImporter`。当来自远程方法调用中的 TPC 需要转换成能够被接收 JBoss 服务器虚拟机使用的事务时，

JBoss 调用该接口。

为安装 JBoss 服务器的事务管理器实现，所有的事务管理服务都需要创建这些 JNDI 绑定。

4.2.2 默认事务管理器

在默认情况下，JBoss 使用快速嵌入虚拟机式的事务管理器。这种事务管理器非常快，但有如下两方面的缺陷。

- 不存在事务日志。因此，服务器瘫痪后，不能够自动恢复。
- 尽管它支持带有远程调用的事务上下文，但不允许将事务上下文传播到其他的虚拟机中。因此，所有的事务工作只能够在 JBoss 服务器驻留的同一 Java 虚拟机中完成。

其中，对应的默认事务管理器 MBean 服务是 `org.jboss.tm.TransactionManagerServiceMBean` 类。它有如下两个属性。

- **TransactionTimeout**: 默认超时时间，以秒计算。默认值为 300 秒，或 5 分钟。
- **XidFactory**: XidFactory 属性指，提供了 `org.jboss.tm.XidFactoryMBean` 实现的 MBean 服务的 JMX ObjectName。XidFactoryMBean 接口用于创建 `javax.transaction.xa.Xid` 实例。某些 XA JDBC 驱动仅仅能够和它们自己的 Xid 实现配合使用，比如旧版本的 Oracle XA 驱动。如果没有指定，系统将使用 JBoss 提供的 Xid 接口实现。

`org.jboss.tm.XidFactory`

XidFactory MBean 是工厂类，即用于创建以 `org.jboss.tm.XidImpl` 形式存在的 `javax.transaction.xa.Xid` 实例。XidFactory 允许通过如下一些属性定制 XidImpl。

- **BaseGlobalId**: 用于创建全局唯一的事务标识。如果多个 JBoss 实例运行在同一 JVM 上，则它必须单独设置。其默认值为 JBoss 服务器的主机名，并追加一斜线。
- **GlobalIdNumber**: 用长整型作为初始事务 id。默认值为 0。
- **Pad**: 它决定了 Xid `getGlobalTransactionId` 和 `getBranchQualifier` 方法返回的 `byte[]` 应该等于最大的 64 位长度，还是小于等于 64 位长度。某些资源管理器，比如 Oracle，要求 id 必须是最大长度。

4.2.3 UserTransaction 支持

JTA `javax.transaction.UserTransaction` 接口允许应用显式地控制事务。对于容器管理事务（BMT）的企业会话 Bean 而言，通过对 Bean 的上下文对象 `javax.ejb.SessionContext` 调用 `getUserTransaction` 方法能够获得 `UserTransaction`。



对于 BMT Bean，不要通过 JNDI 查找获得 `UserTransaction` 接口。这样将违反 EJB 规范。同时，获得的 `UserTransaction` 对象没有钩子，而这些钩子是 EJB 容器用来进行重要检查的依据。

如果打算在其他地方使用 UserTransaction 接口,则必须配置和启动 `org.jboss.tm.usertx.server.ClientUserTransactionService` MBean。该 MBean 在 JNDI 名 “UserTransaction” 下发布了一个 UserTransaction 实现。JBoss 默认情况下配置了这个无配置属性的 MBean。

当单独客户（比如运行在服务器虚拟机之外的客户操作）通过 JNDI 查找以获得 UserTransaction 时,适用于瘦客户的、非常简单的 UserTransaction 将返回给它。UserTransaction 实现仅仅控制该 UserTransaction 对象所在的服务器事务。在客户端完成的本地事务性任务并不包括在 UserTransaction 对象所触发的事务中。

在 JBoss 运行的 JVM 中,通过查找 JNDI 名 “UserTransaction” 获得 UserTransaction 对象时, JTA TransactionManager 的简单接口将返回给调用者。这种情况适合于嵌入并运行在 JBoss 中的 Web 容器上的 Web 组件。当组件部署在嵌入式 Web 服务器中,部署者将把标准的 “`java:/comp/UserTransaction`” 环境命名上下文 (ENC) 名连接到全局的 “UserTransaction”,使得 Web 组件能够遵循 J2EE 规范所给出的 JNDI 名来查找 UserTransaction 实例。

这些代理是基于 EJBModule 中的 invoker-proxy-bindings 元数据创建的。与此同时，单个 EJB 可以和多个代理工厂关联。本文后续内容将给出相关介绍。

(3) ProxyFactory 构建逻辑代理，并将 EJBHome 接口绑定到 JNDI 中。其中，逻辑代理由动态 Proxy (java.lang.reflect.Proxy)、代理暴露的 EJBHome 接口、ClientContainer (org.jboss.proxy.ClientContainer) 形式的 ProxyHandler (java.lang.reflect.InvocationHandler) 实现及客户端拦截器组成。

(4) EJBProxyFactory 创建的代理属于 JDK 1.3 版本及其后续版本的动态代理类型。它是序列化对象，其代理了 EJBModule 元数据中定义的 EJBHome 和远程接口。其中，该代理使用 ClientContainer 处理器将强类型化 EJB 接口创建的请求转换成弱类型化调用。动态代理实例被绑定到 JNDI，并将其作为客户查找的 EJBHome 接口。当客户查找 EJBHome 时，将会传递 Home 代理、ClientContainer 及它的拦截器到客户 JVM 中。因此，使用动态代理避免了其他许多 EJB 容器所需的 EJB 具体编译步骤。

(5) EJBHome 接口是在 ejb-jar.xml 描述符中声明的，并且通过 EJBModule 元数据获得 EJBHome 接口。其中，动态代理的一项重要特性在于它被看成是实现了动态代理所暴露的接口。从 Java 强类型系统的角度考虑，这也是有道理的。代理能够被造型到任何 Home 接口，开发者通过对代理进行反射操作获得接口的所有细节。

(6) 代理将接口触发的调用委派给 ClientContainer 处理器。其中，该处理器只需要实现单个方法：

```
public Object invoke(Object proxy, Method m, Object[] args)
    throws Throwable
```

EJBProxyFactory 创建 ClientContainer，并将它作为 ProxyHandler 看待。与此同时，InvocationContext (org.jboss.invocation.InvocationContext) 和拦截器 (org.jboss.proxy.Interceptor) 链组成了 ClientContainer 的状态信息。其中，InvocationContext 含有如下内容。

- Proxy 关联的 EJB 容器的 MBean JMX ObjectName
- 用于 EJB 的 javax.ejb.EJBMetaData
- EJBHome 接口的 JNDI 名
- 具体传输 Invoker (org.jboss.invocation.Invoker)

其中，拦截器链由构成 EJBHome 或远程接口行为的功能单元组成。这是 EJB 的可配置构件，本文在讨论 jboss.xml 描述符时将会涉及到相关内容介绍。EJBModule 元数据中包含了拦截器构成。拦截器 (org.jboss.proxy.Interceptor) 处理不同类型的 EJB、安全性、事务及传输。开发者也可以添加自己开发的拦截器。

(7) 具体传输 Invoker 和服务器端分离式 Invoker，即与处理 EJB 方法调用传输细节的 Invoker 有千丝万缕的关系。其中，这里的分离式 Invoker 是 JBoss 服务器端组件。

开发者使用 jboss.xml 描述符中的 client-interceptors 元素能够完成客户端拦截器的配置。图 5-2 给出了 jboss.xml DTD 中的客户端拦截器部分。当调用 ClientContainer 的 invoke 方法时，将会创建未类型化 (untyped) 的 Invocation (org.jboss.invocation.Invocation) 实例以封装调用请求。然后，将其传入到拦截器链中。其中，位于拦截器链的最后一个拦截器是传输处理器，即它知道如何将请求发送到服务器，并获得服务器响应，因此它特别关

注具体的传输细节。

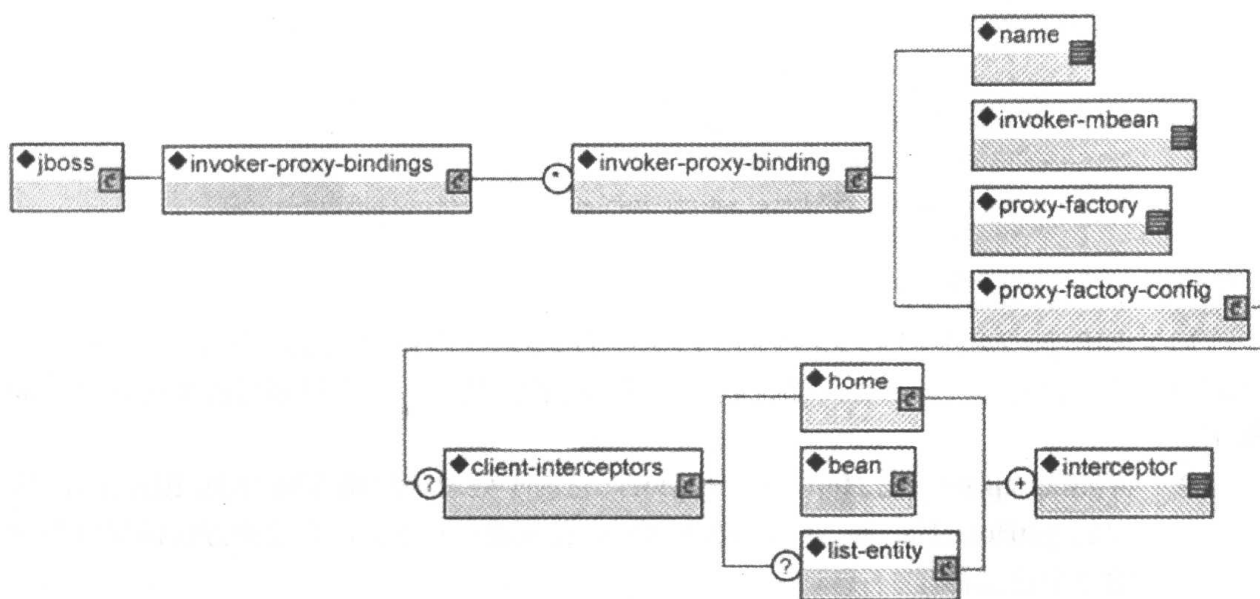


图 5-2 jboss.xml 描述符中的客户端拦截器配置元素

为解释客户端拦截器配置的用法，本文以 server/default/standardjboss.xml 中的默认无状态会话 Bean 配置为例展开如下研究。列表 5-1 给出了“Standard Stateless SessionBean”引用的“stateless-rmi-invoker”客户端拦截器配置。

列表 5-1 “Standard Stateless SessionBean”配置引用的“stateless-rmi-invoker”客户端拦截器

```

<invoker-proxy-bindings>
  <invoker-proxy-binding>
    <name>stateless-rmi-invoker</name>
    <invoker-mbean>jboss:service=invoker,type=jmnp</invoker-mbean>
    <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
    <proxy-factory-config>
      <client-interceptors>
        <home>
          <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
          <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
          <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
          <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
        </home>
        <bean>
          <interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor</interceptor>
          <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
          <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
          <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
        </bean>
      </client-interceptors>
    </proxy-factory-config>
  </invoker-proxy-binding>

```



```
...
<invoker-proxy-bindings>

<container-configuration>
<container-name>Standard Stateless SessionBean</container-name>
<call-logging>false</call-logging>
<invoker-proxy-binding-name>stateless-rmi-invoker</invoker-proxy-binding-name>
...
</container-configuration>
```

如果 EJB jar META-INF/jboss.xml 配置没有覆盖这些拦截器配置，则 JBoss 服务器将其作为无状态会话 Bean 的客户端拦截器的默认配置。其中，各个拦截器提供的具体功能如下：

- **org.jboss.proxy.ejb.HomeInterceptor**，它用于处理客户端 VM 本地 EJBHome 接口的 getHomeHandle、getEJBMetaData 及 remove 方法。其他的方法调用将委派给下个拦截器。
- **org.jboss.proxy.ejb.StatelessSessionInterceptor**，它用于处理客户端 VM 本地 EJBObject 接口的 toString、equals、hashCode、getHandle、getEJBHome 及 isIdentical 方法。其他的方法调用将委派给下个拦截器。
- **org.jboss.proxy.SecurityInterceptor**，它关联方法调用的当前安全上下文，供其他拦截器或服务使用。
- **org.jboss.proxy.TransactionInterceptor**，它关联方法调用的任何活动事务，供其他拦截器使用。
- **org.jboss.invocation.InvokerInterceptor**，它负责封装方法调用的分发，即能够调用正确的具体传输 Invoker。如果客户和服务运行在同一 JVM 中，则该拦截器将通过传址调用以优化调用路由。如果客户运行在服务器 JVM 外，则该拦截器将会把该调用委派给与调用上下文关联的传输 Invoker。比如，对于列表 5-1 给出的配置情形，具体传输 Invoker 将会是与“jboss:service=invoker,type=jrmp”关联的 Invoker 存根。在 2.7.2 小节的“1. JRMPInvoker-RMI/JRMP 传输”中有关于 JRMPInvoker 服务的介绍。

指定 EJB 代理配置

为指定 EJB 调用传输和客户端代理拦截器栈，开发者需要在 EJB jar 中的 META-INF/jboss.xml 描述符，或者在服务器 standardjboss.xml 描述符中定义 invoker-proxy-binding。其中，在 standardjboss.xml 描述符中定义了几个默认的 invoker-proxy-binding 配置，以用于不同的默认 EJB 容器配置和标准的 RMI/JRMP、RMI/IIOP 传输协议。当前，默认的代理配置有：

- **entity-rmi-invoker**：用于实体 Bean 的 RMI/JRMP 配置。
- **clustered-entity-rmi-invoker**：用于群集中实体 Bean 的 RMI/JRMP 配置。
- **stateless-rmi-invoker**：用于无状态会话 Bean 的 RMI/JRMP 配置。

- clustered-stateless-rmi-invoker: 用于群集中无状态会话 Bean 的 RMI/JRMP 配置。
- stateful-rmi-invoker: 用于有状态会话 Bean 的 RMI/JRMP 配置。
- clustered-stateful-rmi-invoker: 用于群集中有状态会话 Bean 的 RMI/JRMP 配置。
- message-driven-bean: 用于消息驱动 Bean 的 JMS Invoker。
- iiop: 会话和实体 Bean 使用的 RMI/IIOP。

为引入新的协议绑定、自定义代理工厂、客户端拦截器栈，开发者需要定义新的 invoker-proxy-binding。图 5-3 给出了 invoker-proxy-binding 的完整 DTD 片段。

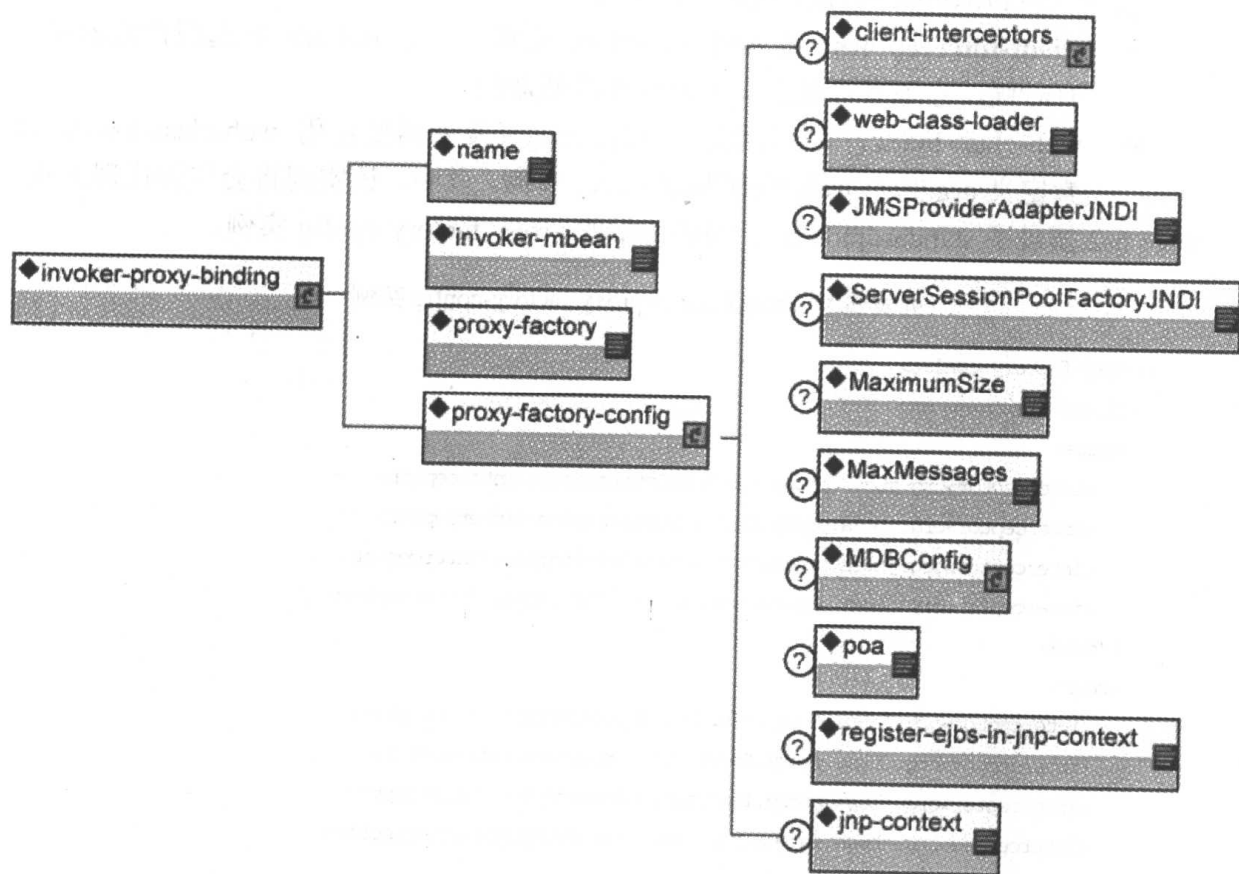


图 5-3 invoker-proxy-binding 内容模型

invoker-proxy-binding 包含如下子元素。

- **name:** name 元素定义了 invoker-proxy-binding 的惟一名。当 EJB 容器配置在设置默认代理绑定和 EJB 部署级别指定的其他代理绑定时，需要引用该名。另外，本文在浏览控制服务器端 jboss.xml 中 EJB 容器配置的相关元素时会有其详细阐述。
- **invoke-mbean:** invoker-mbean 元素给出代理 Invoker 所关联分离式 Invoker MBean 服务的 JMX ObjectName 字符串。
- **proxy-factory:** proxy-factory 元素指定代理工厂的全限定名。其中，该代理工厂必须实现 org.jboss.ejb.EJBProxyFactory 接口。EJBProxyFactory 处理代理配置及其具体协议 Invoker 和上下文。当前，JBoss 提供了如下 EJBProxyFactory 接口实现：
 - org.jboss.proxy.ejb.ProxyFactory: 具体 RMI/JRMP 工厂。
 - org.jboss.proxy.ejb.ProxyFactoryHA: 具体群集 RMI/JRMP 工厂。

- `org.jboss.ejb.plugins.jms.JMSContainerInvoker`: 具体 JMS 工厂。
- `org.jboss.proxy.ejb.IORFactory`: 具体 RMI/IIOP 工厂。
- **proxy-factory-config**: `proxy-factory-config` 元素为 `proxy-factory` 实现提供其他信息。很不幸的是，它当前不是元素的结构化集合。图 5-3 给出的只是应用各类型代理工厂的部分元素。其子元素划分为如下三种调用协议：RMI/JRMP、RMI/IIOP 及 JMS。

对于具体 RMI/JRMP 代理工厂，即 `org.jboss.proxy.ejb.ProxyFactory` 和 `org.jboss.proxy.ejb.ProxyFactoryHA` 相应的元素如下。

- **client-interceptors**: `client-interceptors` 元素定义 EJBHome 和远程拦截器栈。有时候，它还有可能定义多值代理拦截器栈。
 - **web-class-loader**: 为实现动态类装载，开发者需要使用 `web-class-loader` 元素定义 `org.jboss.web.WebClassLoader` 实例。其中，该实例将会与该代理关联。
- 列表 5-2 摘录了 `standardjboss.xml` 描述符中的 `proxy-factory-config` 实例。

列表 5-2 ProxyFactory proxy-factory-config 实例

```
<proxy-factory-config>
  <client-interceptors>
    <home>
      <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
      <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
      <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
      <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
    </home>
    <bean>
      <interceptor>org.jboss.proxy.ejb.EntityInterceptor</interceptor>
      <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
      <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
      <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
    </bean>
    <list-entity>
      <interceptor>org.jboss.proxy.ejb.ListEntityInterceptor</interceptor>
      <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
      <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
      <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
    </list-entity>
  </client-interceptors>
</proxy-factory-config>
```

对于具体 RMI/IIOP 代理工厂，即 `org.jboss.proxy.ejb.IORFactory`，其相应的元素如下。

- **poa**: `poa` 元素定义了便携的对象适配器，或者自用，或者共享使用。
- **register-ejbs-in-jnp-context**: 标志位，表明 EJB 是否应该注册在 JNDI 中。
- **jnp-context**: 用于注册 EJB 的 JNDI 上下文。

➤ **web-class-loader:** 为实现动态类装载, 开发者需要使用 web-class-loader 元素定义 org.jboss.web.WebClassLoader 实例。其中, 该实例将会与该代理关联。列表 5-3 摘录了 standardjboss.xml 描述符中的 proxy-factory-config 实例。

列表 5-3 IOFactory proxy-factory-config 实例

```
<proxy-factory-config>
  <web-class-loader>org.jboss.iiop.WebCL</web-class-loader>
  <poa>per-servant</poa>
  <register-ejbs-in-jnp-context>true</register-ejbs-in-jnp-context>
  <jnp-context>iiop</jnp-context>
</proxy-factory-config>
```

对于具体 JMS 代理工厂, 即 org.jboss.ejb.plugins.jms.JMSContainerInvoker, 其相应的元素如下。

- **MaximumSize:** MaximumSize 元素指明并发 MDB 的最大数量, 它是对于与 JMS 目的地关联的已部署 MDB 而言的。默认情况下, 该元素取值为 15。
- **MaxMessages:** MaxMessages 元素指定, createConnectionConsumer 方法和 createDurableConnectionConsumer 方法的 maxMessages 参数值。其中, createConnectionConsumer 是 javax.jms.QueueConnection 和 javax.jms.TopicConnection 的方法; createDurableConnectionConsumer 是 javax.jms.TopicConnection 的方法。该元素的含义为, 每次与服务器会话的最大消息数量。默认值为 1。如果 JMS 供应商不支持默认值的修改, 则开发者不应该更改其值。
- **MDBConfig:** 配置 MDB JMS 连接行为。其包括如下元素:
 - ✓ **ReconnectIntervalSec:** 重新连接到 JMS 服务器, 而需要等待的时间 (单位: 秒)。
 - ✓ **DLQConfig:** 当重新分发消息次数过多时, 开发者需要使用 DLQConfig 元素配置 MDB 的死信队列 (dead letter queue)。
- **JMSProviderAdaptorJNDI:** JMSProviderAdaptorJNDI 元素配置, 位于 java:/命名空间中 JMS 供应商适配器的 JNDI 名。开发者必须为 MDB 提供该名, 而且必须实现 org.jboss.jms.jndi.JMSProviderAdapter。
- **ServerSessionPoolFactoryJNDI:** ServerSessionPoolFactoryJNDI 元素配置, 位于 JMS 供应商会话池工厂中, java:/命名空间下会话池的 JNDI 名。开发者必须为 MDB 提供该 JNDI 名, 而且必须实现 org.jboss.jms.asf.ServerSessionPoolFactory。

列表 5-4 摘录了 standardjboss.xml 描述符中的 proxy-factory-config 实例。

列表 5-4 JMSContainerInvoker proxy-factory-config 实例

```
<proxy-factory-config>
  <JMSProviderAdapterJNDI>DefaultJMSProvider</JMSProviderAdapterJNDI>
  <ServerSessionPoolFactoryJNDI>StdJMSPool</ServerSessionPoolFactoryJNDI>
  <MaximumSize>15</MaximumSize>
```

```
<MaxMessages>1</MaxMessages>
<MDBConfig>
  <ReconnectIntervalSec>10</ReconnectIntervalSec>
  <DLQConfig>
    <DestinationQueue>queue/DLQ</DestinationQueue>
    <MaxTimesRedelivered>10</MaxTimesRedelivered>
    <TimeToLive>0</TimeToLive>
  </DLQConfig>
</MDBConfig>
</proxy-factory-config>
```

5.2 EJB 服务器端视图

无论如何，各个 EJB 调用必须在驻留了 EJB 容器的 JBoss 服务器中结束。本节内容阐述如何将调用传输给运行 JBoss 服务器的 JVM，并且研究如何借助于 JMX 总线获得其到达 EJB 容器的路径。

分离式 Invoker——传输中间人

在“2.7 远程访问服务——分离式 Invoker”节的内容讨论了分离式 Invoker 架构，并且阐述了如何暴露 MBean 服务的 RMI 兼容接口。通过本章内容，开发者将掌握如下内容，即如何使用分离式 Invoker 暴露 EJB 容器中 Home 和 Bean 接口给客户端。图 5-4 给出了该 Invoker 架构的概要性视图。

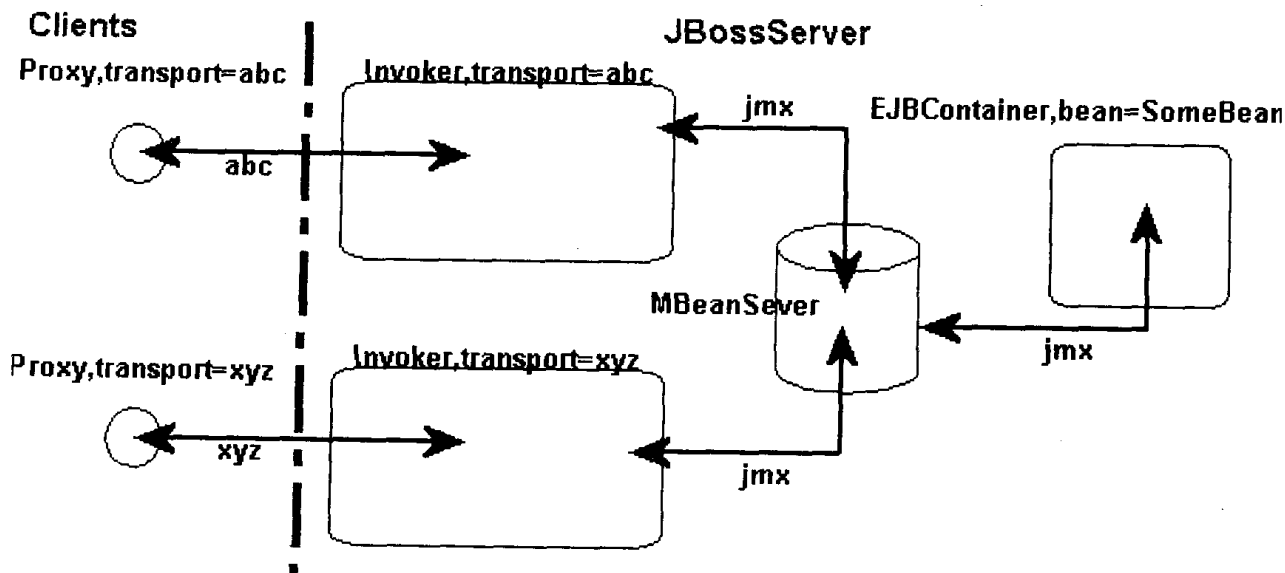


图 5-4 传输 Invoker 服务器端架构

各种 Home 代理都绑定到 Invoker 及其相关传输协议上。在 JBoss 3.2.x 发布版中，容器可能会同时激活多个调用协议。但是对于 JBoss 3.0.x 而言，容器只能激活单个协议，但对具体调用协议不做限定。图 5-5 给出了 jboss.xml DTD 配置中的 Invoker 片段。invoker-

proxy-binding-name 映射到 invoker-proxy-binding/name 元素。在容器配置级，部署于容器中的 EJB 将使用其指定的默认 Invoker。在企业级 Bean 中，invoker-bindings 指定供 EJB 容器 MBean 使用的单个或多个 Invoker。

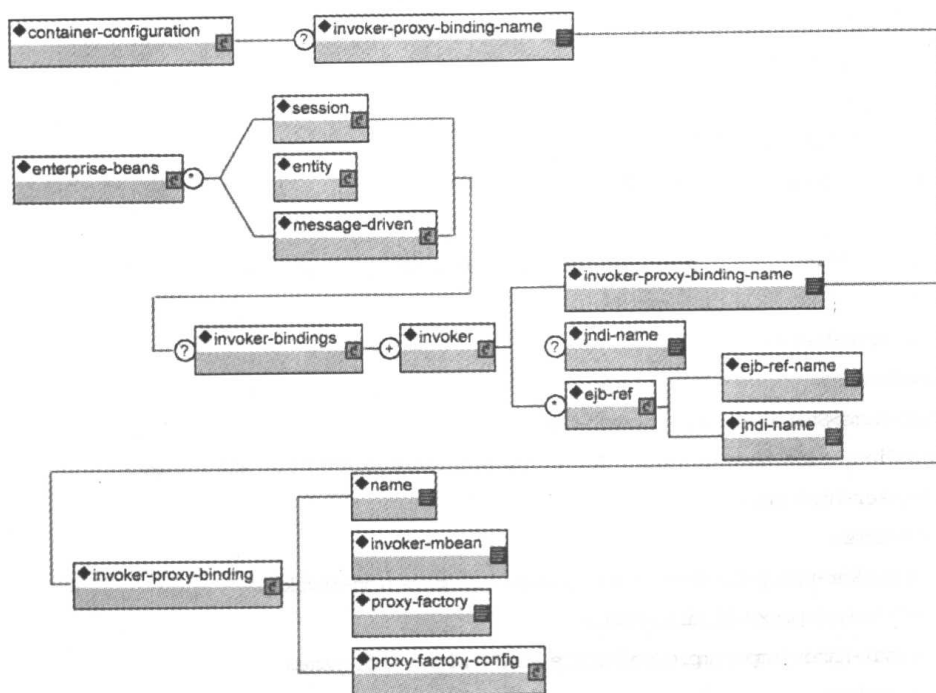


图 5-5 jboss.xml 描述符 Invoker 配置元素

当为某具体 EJB 部署应用指定多个 Invoker 时，开发者必须为 Home 代理定义惟一的 JNDI 绑定位置。其中，开发者通过使用 invoker/jndi-name 元素能够达到此目的。但是，还是存在另外一个问题，即当 EJB 调用其他 EJB 时，这些 Invoker 是如何处理它们所获得的 Home 或远程接口呢？当客户通过代理触发外部 EJB 的依次调用时，为保证返回 Home 和远程接口与该代理的兼容性，需要确保这些接口使用了同一 Invoker。为使 ejb-ref 引用的目标 EJBHome 能够匹配到引用 Invoker 类型，invoker/ejb-ref 元素能够将协议无关的 ENC ejb-ref 映射到 Home 代理绑定。

测试套件中的 org.jboss.test.jrmp 包含有如何使用自定义 JRMPInvoker MBean 的实例。列表 5-5 给出了 JRMPInvoker 配置及其到无状态会话 Bean 的映射。

列表 5-5 为会话 Bean 提供压缩 Socket 的自定义 JRMPInvoker 实例

```
// The custom JRMPInvoker jboss-service.xml descriptor
<server>
  <mbean code="org.jboss.invocation.jrmp.server.JRMPInvoker"
    name="jboss:service=invoker,type=jrmp,socketType=CompressionSocketFactory">
    <attribute name="RMIObjectPort">4445</attribute>
    <attribute name="RMIClientSocketFactory">
      org.jboss.test.jrmp.ejb.CompressionClientSocketFactory
    </attribute>
    <attribute name="RMIServerSocketFactory">
      org.jboss.test.jrmp.ejb.CompressionServerSocketFactory
    </attribute>
  </mbean>
</server>
```



```
</attribute>
</mbean>
</server>

// The jboss.xml descriptor using the custom invoker
<?xml version="1.0"?>
<!DOCTYPE jboss PUBLIC
"-//JBoss//DTD JBOSS 3.2//EN"
"http://www.jboss.org/j2ee/dtd/jboss_3_2.dtd">

<!-- The jboss.xml descriptor for the jmp-comp.jar ejb unit -->
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>StatelessSession</ejb-name>
      <configuration-name>Standard Stateless SessionBean</configuration-name>
      <invoker-bindings>
        <invoker>
          <invoker-proxy-binding-name>stateless-compression-invoker
          </invoker-proxy-binding-name>
          <jndi-name>jmp-compressed/StatelessSession</jndi-name>
        </invoker>
      </invoker-bindings>
    </session>
  </enterprise-beans>

  <invoker-proxy-bindings>
    <invoker-proxy-binding>
      <name>stateless-compression-invoker</name>
      <invoker-mbean>
        jboss:service=invoker,type=jmp,socketType=CompressionSocketFactory
      </invoker-mbean>
      <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
      <proxy-factory-config>
        <client-interceptors>
          <home>
            <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
            <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
            <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
            <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
          </home>
        </bean>
        <interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor</interceptor>
        <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
        <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
        <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
      </proxy-factory-config>
    </invoker-proxy-binding>
  </invoker-proxy-bindings>

```



```

    </bean>
  </client-interceptors>
</proxy-factory-config>
</invoker-proxy-binding>
</invoker-proxy-bindings>
</jboss>

```

其中，默认的 JRMPInvoker 绑定到端口 4445，并且使用了自定义 Socket 工厂，以实现传输层的压缩。另外，StatelessSession EJB 的 invoker-bindings 配置信息指定了“stateless-compression- invoker”，以便绑定在 JNDI 名中，即“jrmc-compressed/Stateless Session”下的 Home 接口将使用它。

有关 HttpInvoker 借助于 RMI/HTTP 协议配置无状态会话 Bean 的实例，开发者可以在测试套件中的 org.jboss.test.hello 包中找到。列表 5-6 给出了这些自定义设置的实例。

列表 5-6 使无状态会话 Bean 使用 RMI/HTTP 的 jboss.xml 描述符实例

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss PUBLIC
  "-//JBoss//DTD JBOSS 3.2//EN"
  "http://www.jboss.org/j2ee/dtd/jboss_3_2.dtd">
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>HelloWorldViaHTTP</ejb-name>
      <jndi-name>helloworld/HelloHTTP</jndi-name>
      <invoker-bindings>
        <invoker>
          <invoker-proxy-binding-name>stateless-http-invoker</invoker-proxy-binding-name>
        </invoker>
      </invoker-bindings>
    </session>
  </enterprise-beans>

  <invoker-proxy-bindings>
    <!-- A custom invoker for RMI/HTTP -->
    <invoker-proxy-binding>
      <name>stateless-http-invoker</name>
      <invoker-mbean>jboss:service=invoker,type=http</invoker-mbean>
      <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
      <proxy-factory-config>
        <client-interceptors>
          <home>
            <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
            <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
            <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
            <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
          </home>

```

```
<bean>
  <interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor</interceptor>
  <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
  <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
  <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
</bean>
</client-interceptors>
</proxy-factory-config>
</invoker-proxy-binding>
</invoker-proxy-bindings>
</jboss>
```

上述内容定义了名为“stateless-http-invoker”的 invoker-proxy-binding。invoker-proxy-binding 使用 HttpInvoker MBean 作为分离式 Invoker。其中，“jboss:service=invoker,type=http”名是 HttpInvoker MBean 的默认名，通过 http-invoker.sar/META-INF/jboss-service.xml 描述符能够寻找到该名，其服务描述符片段如下。

```
<!-- The HTTP invoker service configuration
-->
<mbean code="org.jboss.invocation.http.server.HttpInvoker"
  name="jboss:service=invoker,type=http">
  <!-- Use a URL of the form http://<hostname>:8080/invoker/EJBInvokerServlet
  where <hostname> is InetAddress.getHostName value on which the server
  is running.
  -->
  <attribute name="InvokerURLPrefix">http://</attribute>
  <attribute name="InvokerURLSuffix">:8080/invoker/EJBInvokerServlet</attribute>
  <attribute name="UseHostName">true</attribute>
</mbean>
```

客户代理将 EJB 调用内容提交给 HttpInvoker 服务配置中指定的 EJBInvokerServlet URL。

1. HA JRMPInvoker-群集 RMI/JRMP 传输

org.jboss.invocation.jrmp.server.JRMPInvokerHA 服务扩展 JRMPInvoker，从而能够提供群集敏感 Invoker。自从 JBoss-3.0.3 开始，JRMPInvokerHA 已经能够完整地支持 JRMPInvoker 的所有属性。其含义为，对端口、接口及 Socket 传输的自定义绑定同样也适合于群集 RMI/JRMP。有关群集架构和 HA RMI 代理实现的其他信息，请开发者参考 JBoss 群集（JBoss Clustering）文档。

2. HA HttpInvoker-群集 RMI/HTTP 传输

JBoss 3.0.3 扩展了 JBoss-3.0.2 发布版添加的 RMI/HTTP 层，使得它能够用于群集环境中软件调用的负载均衡。添加了具有 HA 能力的、并扩展了 HttpInvoker 的 HA HttpInvoker，借用了 HA-RMI/JRMP 群集的大量功能。

为使用 HA-RMI/HTTP，开发者需要为 EJB 容器配置 HA HttpInvoker。开发者通过使

用描述符 jboss.xml 或者 standardjboss.xml 能够实现此目的。列表 5-7 给出了测试套件包 org.jboss.test.hello 中的无状态会话 Bean，以作为实例。

列表 5-7 使用 HA-RMI/HTTP 的 jboss.xml 无状态会话配置

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>HelloWorldViaClusteredHTTP</ejb-name>
      <jndi-name>helloworld/HelloHA-HTTP</jndi-name>
      <invoker-bindings>
        <invoker>
          <invoker-proxy-binding-name>stateless-httpHA-invoker</invoker-proxy-binding-name>
        </invoker>
      </invoker-bindings>
      <clustered>true</clustered>
    </session>
  </enterprise-beans>

  <invoker-proxy-bindings>
    <invoker-proxy-binding>
      <name>stateless-httpHA-invoker</name>
      <invoker-mbean>jboss:service=invoker,type=httpHA</invoker-mbean>
      <proxy-factory>org.jboss.proxy.ejb.ProxyFactoryHA</proxy-factory>
      <proxy-factory-config>
        <client-interceptors>
          <home>
            <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
            <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
            <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
            <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
          </home>
          <bean>
            <interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor</interceptor>
            <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
            <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
            <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
          </bean>
        </client-interceptors>
      </proxy-factory-config>
    </invoker-proxy-binding>
  </invoker-proxy-bindings>
</jboss>
```

invoker-proxy-binding-name 元素指定的“stateless-httpHA-invoker”引用了“jboss:service=invoker,type=httpHA” Invoker 服务。其中，上述服务配置在 http-invoker.sar/META-INF/jboss-service.xml 描述符中，其默认配置如下。

```
<mbean code="org.jboss.invocation.http.server.HttpInvokerHA"
  name="jboss:service=invoker,type=httpHA">
  <!-- Use a URL of the form
  http://<hostname>:8080/invoker/EJBInvokerHAServlet
  where <hostname> is InetAddress.getHostname value on which the server
  is running.
  -->
  <attribute name="InvokerURLPrefix">http://</attribute>
  <attribute name="InvokerURLSuffix">:8080/invoker/EJBInvokerHAServlet
  </attribute>
  <attribute name="UseHostName">true</attribute>
</mbean>
```

Invoker 代理使用的 URL，即 EJBInvokerHAServlet，是被映射为部署于群集结点的 Servlet。跨越群集的 HttpInvokerHA 实例形成了候选 HTTP URL 集合，以供客户端代理在处理容错和（或）负载均衡中使用。

5.3 EJB 容器

EJB 容器是管理 EJB 特定类的组件。JBoss 为每个部署其上的惟一 EJB 配置提供了 org.jboss.ejb.Container 实例。但是，实际的实例化对象是 Container 子类，其创建的容器实例由 EJBDeployer MBean 管理。

5.3.1 EJBDeployer MBean

org.jboss.ejb.EJBDeployer MBean 负责创建 EJB 容器。如果准备部署某 EJB-jar，EJBDeployer 将创建并初始化所需的 EJB 容器，各种 EJB 类型都将有相应的 EJB 容器。EJBDeployer 的可配置属性如下。

- **VerifyDeployments:** boolean 标志位，表明是否应该运行 EJB 验证器。它能够对部署单元中的 EJB 进行有效性验证，以判断是否遵循 EJB 2.0 规范。如果设置为 true，则启动了 EJB 验证器。
- **VerifierVerbose:** boolean 变量，控制验证过程中出现的任何失败和警告信息的详细程度。
- **StrictVerifier:** boolean 标志位，表明生效或失效的严格验证。当生效了严格验证时，只有当验证器没有报告错误信息时，EJBDeployer 才会部署 EJB。
- **ValidateDTDs:** boolean 标志位，表明是否借助于声明的 DTD 来验证 ejb-jar.xml 和 jboss.xml 描述符。如果设置为 true，则能确保部署描述符的有效性。
- **MetricsEnabled:** boolean 标志位，用于控制标明 metricsEnabled=true 属性的容器

拦截器是否应该包括在该配置中。其中,它使得开发者能够定义那些包括了 metrics 类型拦截器的容器拦截器配置能够打开或关上。

- **WebServiceName:** WebServiceName 属性指定 Web 服务 MBean 的 JMX ObjectName 字符串。其中,该 MBean 为实现 EJB 类的动态类装载提供了支持。
- **TransactionManagerServiceName:** TransactionManagerServiceName 属性指定, JTA 事务管理器服务的 JMX ObjectName 字符串。其中,开发者必须提供名为 “TransactionManager” 的属性,以返回 javax.transaction.TransactionManager 实例。

该部署器包含了两个核心方法: deploy 和 undeploy。其中, deploy 方法带有一个 URL, 指向 EJB-jar 和目录。对于目录而言,其结构同有效 EJB-jar 相同(主要是为开发过程提供便利而使用)。一旦部署了某应用,对同一个 URL 调用 undeploy 方法便能够卸载它。对已部署 URL 调用 deploy 方法会触发 undeploy 操作,而且该 undeploy 操作将使用同一个 URL,比如重新部署。JBoss 为实现类和接口类的重新部署提供了完整的支持,并能够重新装载那些变化的类。这使得开发者在不用停止正在运行的服务器的情况下,也能够开发和更新 EJB。

EJBDeployer 在部署 EJB jar 及其相关类时,部署器完成如下几方面的主要任务:验证 EJB、为各个不同 EJB 创建容器及使用部署配置信息初始化容器。接下来,本文带领开发者来仔细研究各个步骤。

1. 验证 EJB 部署

当将 EJBDeployer 的 VerifyDeployments 属性设置为 true 时,部署器完成部署包中的 EJB 验证。在验证过程中检查 EJB 组件是否满足 EJB 规范的兼容性要求。其中,它必须保证 EJB 部署单元含有所要求的 Home 和远程、local-home 和本地接口、出现在接口中的对象类型是正确的及实现类中含有所要求的方法。由于 EJB 开发者和部署人员需要经过许多步骤以确保生成的 EJB jar 是正确的,且易于出错。因此,在默认情况下,这是很有用的行为。在验证阶段捕捉到的任何错误和失败信息都能够告诉他们需要纠正的具体错误。

在开发 EJB 过程中,最容易导致问题的地方很可能是如下方面:企业 Bean 实现、远程接口、Home 接口及部署描述符配置的非连接性。很容易导致这些单独元素的不同步。XDoclet 是能够帮助消除该问题的工具,它扩展了标准的 JavaDoc Doclet 引擎。其工作机制为:在 EJB Bean 实现类中添加自定义 JavaDoc 标签,基于此创建远程接口、本地接口及部署描述符。其他详细信息,请开发者参考 XDoclet 主页 <http://sourceforge.net/projects/xdoclet>。

2. 部署 EJB 到容器中

EJBDeployer 最重要的任务是创建 EJB 容器,并将 EJB 组件部署到容器中。部署阶段由迭代 EJB jar 中的 EJB、抽取 ejb-jar.xml 和 jboss.xml 部署描述符中的 Bean 类和相应的元数据组成。对于 EJB jar 中的各个 EJB,需要经历如下步骤:

步骤

- (1) 基于不同的 EJB 类型,创建相应 org.jboss.ejb.Container 的子类。其中 EJB 类型

有：无状态、有状态、BMP 实体、CMP 实体及消息驱动 Bean。同时，JBoss 为容器关联惟一的 ClassLoader，通过它能够装载本地资源。ClassLoader 的惟一性使得它能够与其他 J2EE 组件隔离标准的“java:comp”JNDI 命名空间。

(2) 合并 jboss.xml 和 standardjboss.xml 描述符，以设置所有的容器可配置属性。

(3) 创建和添加为容器配置的容器拦截器。

(4) 将应用对象关联到容器中。其中，这里的应用对象指 J2EE 企业应用，它或许含有多个 EJB 和 Web 上下文。

如果成功部署所有的 EJB，则启动该应用，然后依次启动所有的容器，并使得客户能够使用这些 EJB。如果部署过程中出现任何 EJB 失败，系统会抛出部署异常，随即宣告整个部署模块失败。

3. 容器部署信息

如果所有 EJB 容器配置信息不都是通过遵循 jboss_3_2.dtd 的 XML 文件设置的，则 JBoss 将会通过外部配置来实现它们。图 5-6 给出了 jboss_3_2 DTD 中有关容器配置信息部分。

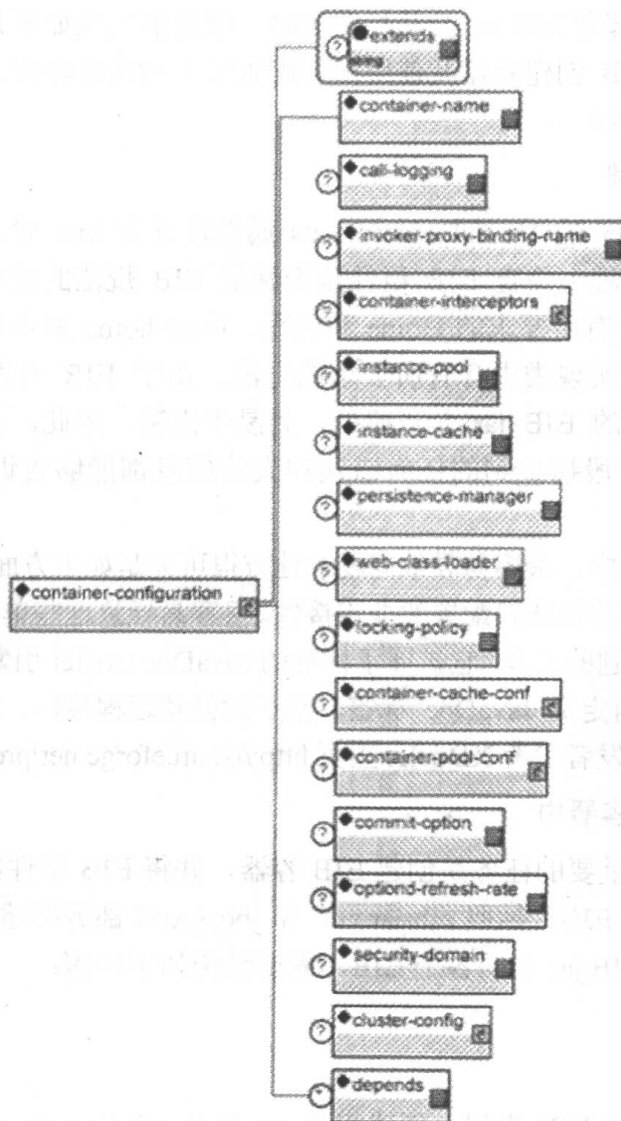


图 5-6 有关容器配置的 jboss_3_2.xml DTD 元素

container-configuration 元素及其子元素为 container-name 元素表明的容器类型指定容器配置信息。每个配置指定大量信息，比如默认 Invoker 类型、容器拦截器组成、实例缓存（池）及其尺寸、持久化管理器、安全性等。因此，存在许多配置信息，这就要求开发者能够深刻理解 JBoss 容器架构。JBoss 发布版为 4 种 EJB 类型提供了标准的配置。其中，配置文件名为 standardjboss.xml，它位于所有使用 EJB 的配置文件集合的 conf 目录下。列表 5-8 给出了 standardjboss.xml 中的配置实例。

列表 5-8 摘自 server/default/conf/standardjboss.xml 文件的一个复杂 container-configuration 元素实例

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss PUBLIC
    "-//JBoss//DTD JBOSS 3.2//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_3_2.dtd">
<jboss>
...
    <container-configuration>
        <container-name>Standard CMP 2.x EntityBean</container-name>
        <call-logging>false</call-logging>
        <invoker-proxy-binding-name>entity-rmi-invoker</invoker-proxy-binding-name>
        <sync-on-commit-only>false</sync-on-commit-only>
        <container-interceptors>
            <interceptor>org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.TxInterceptorCMP</interceptor>
            <interceptor metricsEnabled="true">org.jboss.ejb.plugins.MetricsInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.EntityCreationInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.EntityLockInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.EntityInstanceInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.EntityReentranceInterceptor</interceptor>
            <interceptor>org.jboss.resource.connectionmanager.CachedConnectionInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.EntitySynchronizationInterceptor</interceptor>
            <interceptor>org.jboss.ejb.plugins.cmp.jdbc.JDBCRelationInterceptor</interceptor>
        </container-interceptors>
        <instance-pool>org.jboss.ejb.plugins.EntityInstancePool</instance-pool>
        <instance-cache>org.jboss.ejb.plugins.InvalidableEntityInstanceCache</instancecache>
        <persistence-manager>org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager</persistence-manager>
        <locking-policy>org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLock</lockingpolicy>
        <container-cache-conf>
            <cache-policy>org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy</cachepolicy>
            <cache-policy-conf>
                <min-capacity>50</min-capacity>
                <max-capacity>1000000</max-capacity>
                <overager-period>300</overager-period>
                <max-bean-age>600</max-bean-age>
                <resizer-period>400</resizer-period>
            </cache-policy-conf>
        </container-cache-conf>
    </container-configuration>
</jboss>
```

```
<max-cache-miss-period>60</max-cache-miss-period>
<min-cache-miss-period>1</min-cache-miss-period>
<cache-load-factor>0.75</cache-load-factor>
</cache-policy-conf>
</container-cache-conf>
<container-pool-conf>
  <MaximumSize>100</MaximumSize>
</container-pool-conf>
<commit-option>B</commit-option>
</container-configuration>
...
</container-configurations>
</jboss>
```

图 5-6 和列表 5-8 给出了容器配置选项的具体情况。开发者可以通过两个级别指定容器配置信息。第一，位于配置文件集合目录中的 `standardjboss.xml`（译者注：位于 `conf` 目录）文件。第二，`ejb-jar` 级别。通过 `ejb-jar META-INF` 目录存放 `jboss.xml` 文件，能够覆盖 `standardjboss.xml` 文件中的相应容器配置，或者指定新的容器配置元素。因此，它们为容器配置提供了很大的灵活性。正如开发者所看到的一样，所有的容器配置属性都能够通过外部配置，并易于修改。有经验的开发者甚至能够实现其擅长的容器组件，比如实例池或缓存，并且很容易实现和标准容器配置的集成，从而实现某特定应用或环境的优化行为。

其中，EJB 部署具体选择的部署配置是基于显式或隐式的 `jboss/enterprise-beans/` `<type>/configuration-name` 元素。图 5-7 给出的 `jboss.xml` DTD 片段，展示了 EJB 是如何声明其使用的容器配置。

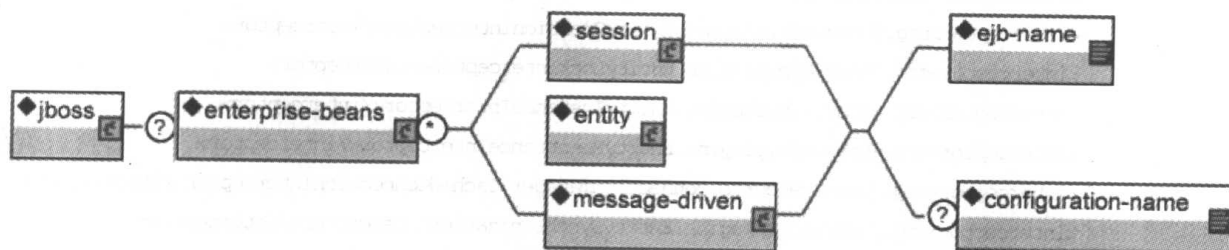


图 5-7 jboss.xml 描述符 EJB 到容器配置的映射

`configuration-name` 元素连到 `container-configurations/container-configuration` 元素，其位于图 5-6 中。它指定了 EJB 使用的容器配置。其中，该连接是从 `configuration-name` 元素指向 `container-name` 元素的。通过在 EJB 定义中包括 `container-configuration` 元素便能够为 EJB 的各个类指定容器配置。一般情况下，尽管 JBoss 支持定义全新的容器配置，但开发者都不会这样做的。通常用法是使用 `jboss.xml` 级别的 `container-configuration` 覆盖 `standardjboss.xml` 描述符中 `container-configuration` 的一个或多个部分。通过指定 `container-configuration`，即引用了 `standardjboss.xml` 中现有的 `container-configuration/container-name`，以作为属性 `container-configuration/extends` 的取值。列表 5-9 定义了新配置“Secured Stateless SessionBean”，其扩展了标准会话 Bean 配置“Standard Stateless SessionBean”。

列表 5-9 覆盖 standardjboss.xml 中“Standard Stateless SessionBean”容器配置以使用保护访问的实例

```
<?xml version="1.0"?>
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>EchoBean</ejb-name>
      <configuration-name>Secured Stateless SessionBean</configuration-name>
    ...
  </session>
</enterprise-beans>
<container-configurations>
  <container-configuration extends="Standard Stateless SessionBean">
    <container-name>Secured Stateless SessionBean</container-name>
    <!-- Override the container security domain -->
    <security-domain>java:/jaas/my-security-domain</security-domain>
  </container-configuration>
</container-configurations>
</jboss>
```

如果 EJB 没有在部署单元 ejb-jar 中提供容器配置规范，则容器工厂会根据 EJB 类型从 standardjboss.xml 描述符中选择一个容器配置。因此，在实际应用中各类型 EJB 都存在隐式 configuration-name 元素。其中，EJB 类型到默认容器配置名的映射关系如下。

- 容器管理持久化实体版本 2.0 = Standard CMP 2.x EntityBean
- 容器管理持久化实体版本 1.1 = Standard CMP EntityBean
- Bean 管理持久化实体 = Standard BMP EntityBean
- 无状态会话 Bean = Standard Stateless SessionBean
- 有状态会话 Bean = Standard Stateful SessionBean
- 消息驱动 Bean = Standard Message Driven Bean

因此，如果开发者打算使用基于 EJB 类型的默认容器配置，则不用给出 EJB 使用的容器配置。另外，开发者也可能借助于独立的描述符来包括 configuration-name 元素，当然这只是使用风格的不同。

至此，开发者已经知道如何制定 EJB 使用的容器配置，并能够定义部署单元级覆盖配置，但问题是：那些 container-configuration 的子元素各自的含义呢？接下来本文将展开论述。其中，很多元素都指定了接口的类实现，而且元素之间有相互作用。因此，本文在深入到配置元素前，首先带领开发者来理解 org.jboss.metadata.XmlLoadable 接口。

XmlLoadable 只是含有单个方法的简单接口。其定义如下。

```
import org.w3c.dom.Element;
public interface XmlLoadable
{
    public void importXml(Element element) throws Exception;
}
```

实现该接口的类能够借助于 XML 文档片段指定各自的配置。传入到 `importXml` 方法的内容是文档片的根元素。接下来的内容将有若干个这方面的容器配置元素实例。

（1）`container-name` 元素

`container-name` 元素为具体配置指定惟一。通过将 EJB configuration-name 取值设成容器配置的 `container-name` 值，EJB 便能连接到特定容器配置。

（2）`call-logging` 元素

`call-logging` 元素为 `boolean` 取值（`true` 或 `false`），其表明 `LogInterceptor` 是否记录方法对容器的调用日志。由于 `Log4j` 提供了细粒度日志功能 API，因此现在基本上没有使用到它。

（3）`invoker-proxy-binding-name` 元素

`invoker-proxy-binding-name` 元素指定使用的默认 `Invoker` 名。如果企业 Bean 级别的 `invoker-bindings` 规范不存在，则匹配 `invoker-proxy-binding-name` 元素取值的 `invoker-proxy-binding` 将被用来创建 `Home` 和远程代理。

（4）`container-interceptors` 元素

`container-interceptors` 元素指定一个或多个 `interceptor` 元素，从而为容器配置方法拦截器链。`interceptor` 元素值为实现了 `org.jboss.ejb.Interceptor` 接口的全限定类名。容器拦截器形成了传递给 EJB 方法调用的连接链表结构。当 `MBeanServer` 传递方法调用给容器时，链中的首个拦截器将受到调用。末尾的拦截器调用企业 Bean 的业务方法。本章在后面讨论容器插件式框架时，将会讨论 `Interceptor` 接口。通常情况下，当开发者需要更改现有的标准 EJB 拦截器配置时，请务必小心。因为 EJB 规范表明，安全性、事务、持久化及线程安全性都是来自于这些拦截器。

（5）`instance-pool` 和 `container-pool-conf` 元素

`instance-pool` 元素，指定实现 `org.jboss.ejb.InstancePool` 接口的全限定类名，以作为容器 `InstancePool` 使用。如图 5-8 所示。本章在后面讨论容器插件式框架时，将会深入讨论 `InstancePool` 接口。

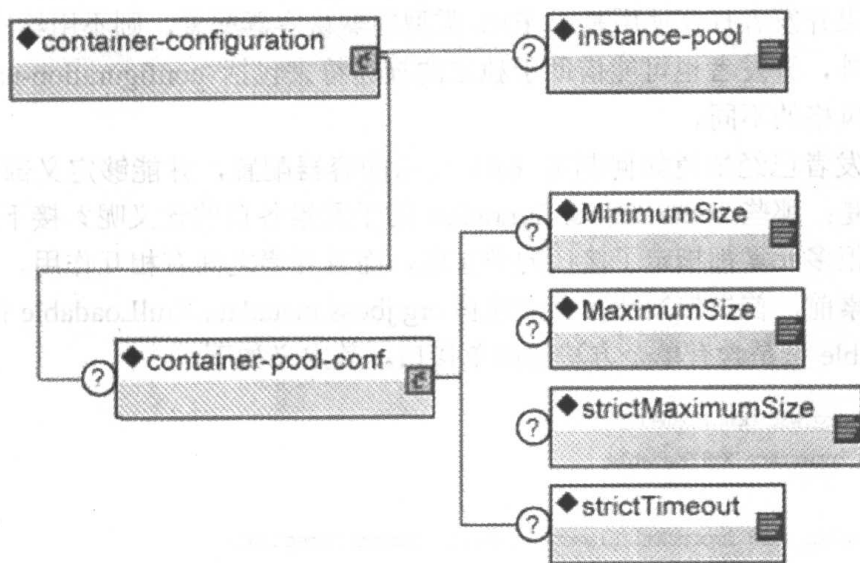


图 5-8 `instance-pool` 和 `container-pool-conf` 元素

如果 instance-pool 元素指定的 InstancePool 实现类实现了 XmlLoadable 接口, container-pool-conf 将被传入到该实现类中。当前, JBoss 所有的 InstancePool 实现都继承于 org.jboss.ejb.plugins.AbstractInstancePool 类, 并且都支持 container-pool-conf 所有的子元素, 即 MinimumSize、MaximumSize、strictMaximumSize 及 strictTimeout。MinimumSize 元素给出了保持在池中的最小实例数量, 但 JBoss 目前并没有将 InstancePool 延伸 (seed) 到 MinimumSize 值。

MaximumSize 指定所允许的最大池实例数量。默认的 MaximumSize 并不能满足要求。当然, 它表明能够保持在池中的最大 EJB 实例数量, 但如果并发请求数目超过了 MaximumSize 值, JBoss 还是会创建所需的 EJB 实例的。如果确实打算限制 EJB 实例的最大并发数量, 则开发者需要将 strictMaximumSize 元素设成 true。一旦 strictMaximumSize 元素设置为 true, 则最多能激活 MaximumSize 个 EJB 实例。如果已经存在 MaximumSize 个激活的实例, 则随后的请求将一直阻塞到实例返还给池为止。strictTimeout 元素控制了阻塞等待实例池对象的时间。它以毫秒定义了当存在 MaximumSize 个缉获的实例时, 等待实例返还给池的时间。小于或等于 0 值都意味着请求不会去等待。当请求等待时间超过 strictTimeout 值后, 系统会抛出 java.rmi.ServerException 异常, 客户从而放弃调用。其默认值为 Long 类型, 即允许的最大等待时间为 9 223 372 036 854 775 807 毫秒, 大约是 292 471 208 年。

(6) instance-cache 和 container-cache-conf 元素

instance-cache 元素, 指定实现 org.jboss.ejb.InstanceCache 接口的全限定类名。该元素仅对实体 Bean 和有状态会话 Bean 有效, 因为只有这两个 EJB 类型才具有关联的标识 (identity), 如图 5-9 所示。本章后面讨论容器插件式框架时, 将会深入讨论 InstanceCache 接口。

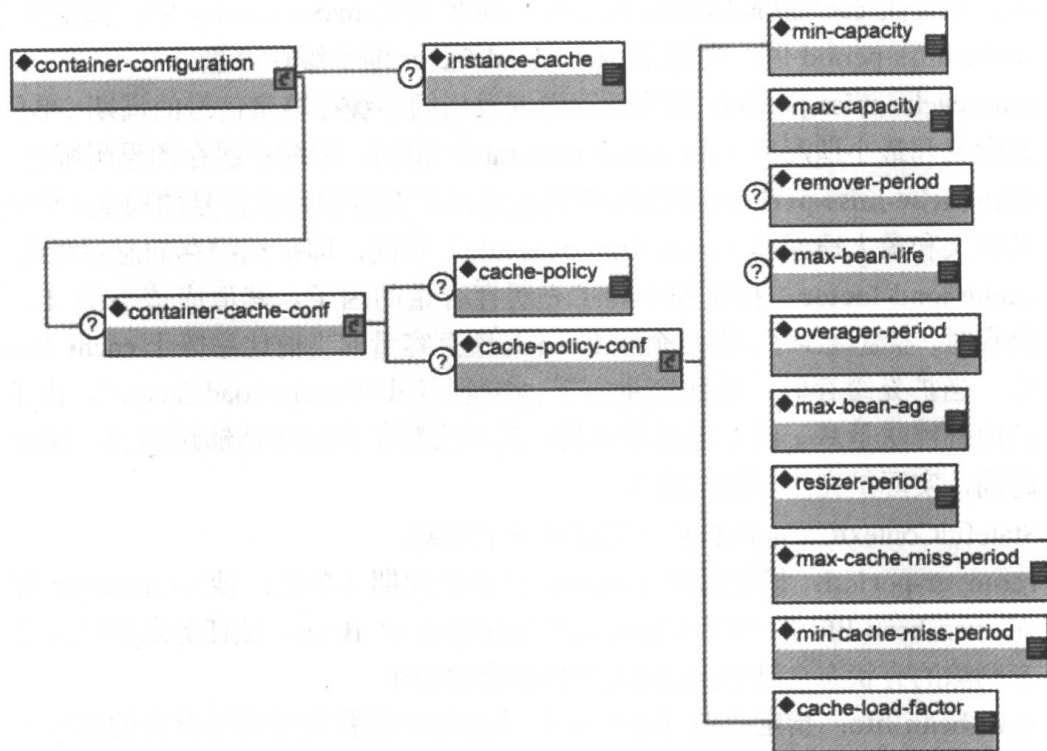


图 5-9 instance-cache、container-cache-conf 以及相关元素

如果 InstanceCache 实现支持 XmlLoadable 接口，container-cache-conf 元素将被传入到该实现类中。当前，JBoss 所有的 InstanceCache 实现都继承于 org.jboss.ejb.plugins.AbstractInstanceCache 类中，并支持 XmlLoadable 接口。同时，它们使用 cache-policy 子元素作为 org.jboss.util.CachePolicy 实现的全限定类名，以用于实例缓存存储源。如果 CachePolicy 实现支持 XmlLoadable 接口，则子元素 cache-policy-conf 将传入到该实现中。如果不支持，则将忽略 cache-policy-conf 子元素。

当前 JBoss 存在两个 CachePolicy 实现，而且这些实现能被 standardjboss.xml 描述符使用。同时，它们支持 cache-policy-conf 子元素。其中，上述两个 CachePolicy 分别是 org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy 和 org.jboss.ejb.plugins.LRUStatefulContextCachePolicy。前一个 CachePolicy 实现用于实体 Bean 容器，而后一个用于有状态会话 Bean 容器。它们都支持如下 cache-policy-conf 子元素。

- **min-capacity:** 指定缓存的最小容量。
- **max-capacity:** 指定缓存的最大容量，其值不能小于 min-capacity。
- **overager-period:** 指定运行 overager 任务的周期（单位：秒）。overager 任务的目的在于查看，缓存是否含有时间超过 max-bean-age 元素值的企业 Bean。任何满足该条件的企业 Bean 都将被挂起（passivate）。
- **max-bean-age:** 指定 overager 任务挂起企业 Bean 之前，它所能处于非活动状态的最大时间（单位：秒）。
- **resizer-period:** 指定运行 resizer 任务的周期（单位：秒）。resizer 任务的目的在于根据如下给出的 3 个元素值缩小或扩充缓存容量。当运行 resizer 任务时，它会检查当前的缓存遗漏（cache miss）周期。如果该周期小于 min-cache-miss-period 值，则使用 cache-load-factor 将缓存容量扩充到 max-capacity 中。如果大于 max-cache-miss-period 值，则使用 cache-load-factor 缩小缓存容量。
- **max-cache-miss-period:** 指定缓存遗漏发出缩小缓存容量信号的周期（单位：秒）。其含义和最小遗漏率（minimum miss rate）相同，即缩小缓存的忍耐限度。
- **min-cache-miss-period:** 指定缓存遗漏发出扩充缓存容量信号的周期（单位：秒）。其含义和最大遗漏率（maximum miss rate）相同，即扩充缓存的忍耐限度。
- **cache-load-factor:** 指定缩小和扩充缓存容量的因子。其值应该小于 1。当缩小缓存时，容量减少了，因此企业 Bean 与缓存容量的当前比率等于 cache-load-factor 值。当扩充缓存时，新的容量等于“当前容量*1/cache-load-factor”。由于 JBoss 的内部算法是基于缓存遗漏数量的，所以实际扩充因子能够达到 2。缓存遗漏率越高，实际扩充因子越接近 2。

LRUStatefulContextCachePolicy 也支持如下子元素。

- **remover-period:** 指定运行 remover 任务的周期（单位：秒）。remover 任务删除在 max-bean-life 秒都没有访问的被挂起的企业 Bean。该任务还能防止那些用户未删除的有状态会话 Bean 装入到挂起存储源中。
- **max-bean-life:** 指定企业 Bean 在从挂起存储源删除前所能存在的最大非活动时间（单位：秒）。

另一种 CachePolicy 实现是 org.jboss.ejb.plugins.NoPassivationCachePolicy 简化类，它

从不挂起实例。它使用内存 HashMap 实现。如果没有显式删除实例，它从不丢弃这些实例。同时，该类也不支持任何 cache-policy-conf 配置元素。

(7) persistence-manager 元素

persistence-manager 元素值指定持久化管理器实现的全限定类名。其中，实现类型取决于 EJB 类型。对于有状态会话 Bean 而言，开发者需要实现 org.jboss.ejb.StatefulSessionPersistenceManager 接口。对于 BMP 实体 Bean 而言，开发者需要实现 org.jboss.ejb.EntityPersistenceManager 接口；而对于 CMP 实体 Bean 而言，开发者则需要实现 org.jboss.ejb.EntityPersistenceStore 接口。

(8) web-class-loader 元素

web-class-loader 元素指定 org.jboss.web.WebClassLoader 的子类。它和 WebServiceMBean 一起实现已部署 EAR、EJB jar 及 WAR 中资源和类的动态装载。其中，WebClassLoader 将和容器关联上，并且将 org.jboss.mx.loading.UnifiedClassLoader 作为其双亲。它重载了 getURLs() 方法，以返回用于远程装载的不同 URL 集合，而不是用于本地装载。

WebClassLoader 提供了两个方法供其子类重载：getKey() 和 getBytes()。后者在本实现中并不具有可操作性，而应该用具备生成字节码能力的子类重载它，比如 iiop 模块使用的类装载器。

WebClassLoader 子类必须提供 WebClassLoader (ObjectName containerName, UnifiedClassLoader parent) 这样的命名方法构建器。

(9) locking-policy 元素

locking-policy 元素给出 EJB 锁实现的全限定类名。其中，该类必须实现 org.jboss.ejb.BeanLock 接口。当前的 JBoss 版本包括如下锁实现：

- **org.jboss.ejb.plugins.lock.MethodOnlyEJBLOCK:** 不具备悲观事务锁功能的实现。它为单线程、不可重入 (non-reentrant) 企业 Bean 提供锁支持。自从 JBoss 3.0.4 开始，JBoss 就不再支持这种锁策略。
- **org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLOCK:** 它持有、等待公平 (fair) FIFO 队列中的事务锁被释放，这样一种线程。当然，非事务性线程也可以放在该等待队列中。该类从队列中弹出下一个等待事务，并通知那些关联该事务的线程。当前，QueuedPessimisticEJBLOCK 是标准配置中使用的默认类。
- **org.jboss.ejb.plugins.lock.SimpleReadWriteEJBLOCK:** 该锁允许并发多个读锁。一旦某些操作请求了该锁，后续那些未获得读锁的读锁请求将一直被阻塞到所有的写操作完成，然后所有等待的读操作将并发执行 (依赖于 reentrant 设置 /methodLock)。如果某读操作先于其他等待写操作初次获得写锁，并且已经有读操作正在执行，则会抛出不一致读异常信息。当然，写操作在获得写锁之前，必须等待所有的读锁都成功释放。
- **org.jboss.ejb.plugins.lock.NoLock:** “Instance Per Transaction *” 容器配置使用的无锁 (anti-locking) 策略。
- **org.jboss.ejb.plugins.lock.JDBCOptimisticLock:** “Instance Per Transaction CMP 2.x EntityBean” 容器配置使用的乐观锁策略。

在“5.4 实体 Bean 锁和死锁检测”节中将详细介绍锁策略。

（10）commit-option 和 optiond-refresh-rate 元素

commit-option 元素值指定 EJB 实体 Bean 持久化存储的提交方式。其取值范围为：A、B、C 或 D。各自具体含义如下。

- A，容器缓存事务之间企业 Bean 的状态。该选项假使容器是访问持久存储源的惟一用户。这使得只有在真正需要同步内存状态和持久存储源时，容器才会理会它。在如下场合才会发生同步操作：执行目标企业 Bean 的第一个业务方法之前，或在企业 Bean 被挂起、并再次激活以服务其他业务方法之后。同时，这种同步行为与业务方法是否运行在事务上下文中无关。
- B，容器缓存事务之间企业 Bean 的状态。但不同于选项 A，它并不假使容器以独占方式访问持久存储源。因此，在每次事务开始时，容器便需要同步内存状态。执行在事务上下文的业务方法并没有从容器缓存企业 Bean 中受益，然而执行在事务上下文（事务属性 Never、NotSupported 或 Supports）外部的业务方法却能访问企业 Bean 的缓存状态（有可能是无效的状态）。
- C，容器并不缓存企业 Bean 实例。每次事务开始时都需要同步内存状态。对于执行在事务之外的业务方法，也会发生同步操作，但是执行 ejbLoad 的事务上下文同调用者的相同。
- D，JBoss 独有的特性，EJB 规范并没有给出。它实现了惰性（lazy）读操作，即和选项 A 一样，缓存事务之间企业 Bean 的状态。但有一点和它不同，即选项 D 会定期重新同步持久源。其同步持久源的默认时间为 30 秒，开发者通过 optiond-refresh-rate 元素能够完成其配置。

（11）security-domain 元素

在 EJB org.jboss.ejb.Container 类内部，security-domain 元素指定对象的 JNDI 名。其中，该对象实现了 org.jboss.security.AuthenticationManager 和 org.jboss.security.RealmMapping 接口。通常情况下，开发者会在 jboss 根元素指定全局 security-domain，使得具体部署中的所有 EJB 都受到保护。“第 8 章 JBoss 之安全性——J2EE 安全性配置和架构”中有关于安全性管理器接口及配置安全性层的深入讨论。

（12）cluster-config 元素

cluster-config 及其相关元素如图 5-10 所示。

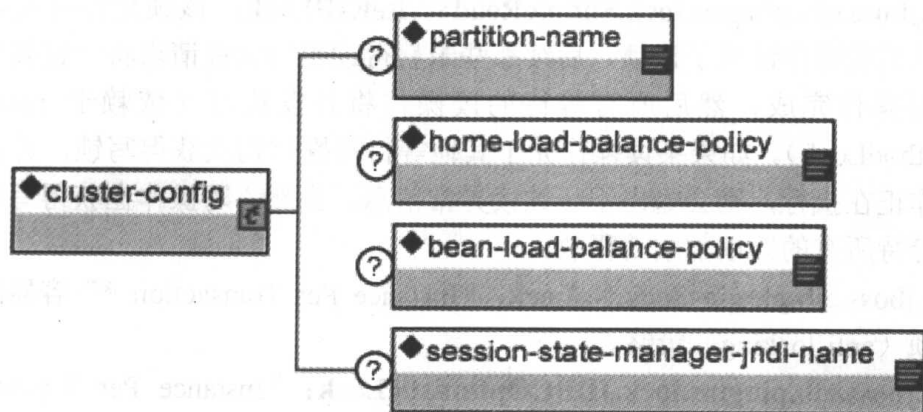


图 5-10 cluster-config 及其相关元素

cluster-config 元素为那些使用容器配置的所有 EJB 指定具体群集配置。开发者可以从两个级别指定群集配置：容器配置级或单个 EJB 部署级。

partition-name 元素指定寻找 org.jboss.ha.framework.interfaces.HAPartition 接口的具体位置，使得容器能够交换群集信息。它并不是绑定在 HAPartition 下的 JNDI 全名，而是对应于管理目标群集的 ClusterPartitionMBean 服务的 PartitionName 属性。通过追加“/HASessionState”到 partition-name 值便构成 HAPartition 绑定的实际 JNDI 名。其默认值为“DefaultPartition”。

home-load-balance-policy 元素指定 Java 类名，以用于对 Home 代理调用的负载均衡。该类必须实现 org.jboss.ha.framework.interface.LoadBalancePolicy 接口。默认值为 org.jboss.ha.framework.interfaces.RoundRobin。

bean-load-balance-policy 元素指定 Java 类名，以用于对 Bean 代理调用的负载均衡。它必须实现 org.jboss.ha.framework.interface.LoadBalancePolicy 接口。其默认值为：

- 对于实体和有状态会话 Bean，org.jboss.ha.framework.interfaces.FirstAvalible。
- 对于无状态会话 Bean，org.jboss.ha.framework.interfaces.RoundRobin。

其中，session-state-manager-jndi-name 元素指定供容器使用的 org.jboss.ha.framework.interfaces.HASessionState 的名字，以用于群集中状态会话管理的后端支持。不同于 partition-name 元素，它是绑定在 HASessionState 实现下的 JNDI 名。其默认位置为“/HASessionState/Default”。

(13) depends 元素

depends 元素给出了容器或 EJB 所依赖服务的 JMX ObjectName。显式地给出其他服务的依赖性约定，即先启动要求的服务，从而避免了对部署顺序的依赖。

5.3.2 容器插件式框架

JBoss 使用框架模式实现了 EJB 容器，其允许更换组成容器行为的各个构件实现。容器本身除了负责将不同行为组件连接在一起外，并没有完成其他的重要工作。行为组件的实现称为插件，比如能够插入新的实现到容器中以更改容器配置。其中，能够变换的插件式行为实例有持久化管理、对象池、对象缓存、容器 Invoker 及拦截器。org.jboss.ejb.Container 类存在 4 个子类，每个子类实现了特定的企业 Bean 类型。

- org.jboss.ejb.EntityContainer 处理 javax.ejb.EntityBean 类型。
- org.jboss.ejb.StatelessSessionContainer 处理无状态 javax.ejb.SessionBean 类型。
- org.jboss.ejb.StatefulSessionContainer 处理有状态 javax.ejb.SessionBean 类型。
- org.jboss.ejb.MessageDrivenContainer 处理 javax.ejb.MessageDrivenBean 类型。

EJB 容器将其大部分行为委派给组件，即容器插件。构成容器插入点的接口如下。

- org.jboss.ejb.ContainerPlugin
- org.jboss.ejb.ContainerInvoker
- org.jboss.ejb.Interceptor
- org.jboss.ejb.InstancePool
- org.jboss.ejb.InstanceCache

- org.jboss.ejb.EntityPersistenceManager
- org.jboss.ejb.EntityPersistenceStore
- org.jboss.ejb.StatefulSessionPersistenceManager

容器的主要职责是管理插件，其意味着插件本身包含了实现各自功能的所有信息。

1. org.jboss.ejb.ContainerPlugin

ContainerPlugin 接口是所有容器插件接口的双亲接口。它提供的回调使得容器能够为各个插件提供插入点，以指向插件工作的容器。列表 5-10 给出了 ContainerPlugin 接口。

列表 5-10 org.jboss.ejb.ContainerPlugin 接口

```
public interface ContainerPlugin extends org.jboss.system.Service
{
    /** This callback is set by the container so that the plugin
     * may access its container
     *
     * @param con the container which owns the plugin
     */
    public void setContainer(Container con);
}
```

2. org.jboss.ejb.Interceptor

开发者能够使用 Interceptor 接口创建方法拦截器链，并且每次 EJB 方法调用都会传递给它们。列表 5-11 给出了 Interceptor 接口。

列表 5-11 org.jboss.ejb.Interceptor 接口

```
import org.jboss.invocation.Invocation;

public interface Interceptor extends ContainerPlugin
{
    public void setNext(Interceptor interceptor);
    public Interceptor getNext();
    public Object invokeHome(Invocation mi) throws Exception;
    public Object invoke(Invocation mi) throws Exception;
}
```

容器配置定义的所有拦截器都将被 EJBDeployer 创建并添加到容器拦截器链中。末尾的拦截器并不是由该部署器添加的，而是由容器本身完成，因为该拦截器需要完成与 EJB Bean 实现进行交互。

链中的拦截器顺序很重要。顺序背后所隐藏的道理是，在拦截器与缓存和池交互前，未绑定到特定 EnterpriseContext 实例的拦截器位置已经确定。

Interceptor 接口的实现者形成了连接链表的类似结构，Invocation 对象将传入到各个实现者中。当 Invoker 借助于 JMX 总线传递 Invocation 给容器时，将会触发对链中首个拦截器的调用。末尾的拦截器将调用企业 Bean 的业务方法。一般情况下，根据企业 Bean 类

型和容器配置的不同,该链中存在的 5 个拦截器依次排开。Interceptor 语义复杂度有简单的,也有复杂的。比如,LoggingInterceptor 是简单拦截器、EntitySynchronizationInterceptor 是复杂拦截器。

拦截器模式的一个主要优势在于拦截器排列的灵活性。另一个优势在于不同拦截器所完成功能的明确划分。比如,事务和安全性逻辑被 TXInterceptor 和 SecurityInterceptor 各自清晰地分开了。

如果任意一个拦截器出现错误,该调用将在此处终止。这被称为快速失败语义类型(fail-quickly type of semantic)。比如,如果没有提供正确的权限访问受保护的 EJB,在事务启动或实例缓存更新前,由于存在 SecurityInterceptor,因此调用将失败。

3. org.jboss.ejb.InstancePool

InstancePool 用于管理未关联任何标识的 EJB 实例。该池实际上是管理 org.jboss.ejb.EnterpriseContext 对象的子类,该对象聚集了未与标识关联的企业 Bean 实例和相关数据。列表 5-12 给出了 InstancePool 接口。

列表 5-12 org.jboss.ejb.InstancePool 接口

```
public interface InstancePool extends ContainerPlugin
{
    /** Get an instance without identity. Can be used
     * by finders and create-methods, or stateless beans
     *
     * @return Context /w instance
     * @exception RemoteException
     */
    public EnterpriseContext get() throws Exception;

    /** Return an anonymous instance after invocation.
     *
     * @param ctx
     */
    public void free(EnterpriseContext ctx);

    /** Discard an anonymous instance after invocation.
     * This is called if the instance should not be reused,
     * perhaps due to some exception being thrown from it.
     *
     * @param ctx
     */
    public void discard(EnterpriseContext ctx);

    /**
     * Return the size of the pool.
     *
     * @return the size of the pool.
     */
}
```

```
public int getCurrentSize();  
/**  
 * Get the maximum size of the pool.  
 *  
 * @return the size of the pool.  
 */  
public int getMaxSize();  
}
```

根据配置的不同，容器可能会选择一定大小的池，以容纳待回收实例，或者在需要时再次实例化和初始它们。

InstanceCache 实现使用池，以用于激活可用实例。拦截器也使用它获得实例，从而为 Home 接口方法（create 和 finder 调用）所使用。

4. org.jboss.ejb.InstanceCache

容器 InstanceCache 实现处理处于激活状态的所有 EJB 实例，即那些关联标识的企业 Bean 实例。只有实体 Bean 和有状态会话 Bean 才会被缓存，因为只有这些企业 Bean 类型才具有方法调用之间的状态信息。其中，实体 Bean 的缓存键是企业 Bean 的主键。有状态会话 Bean 的缓存键是会话 id。列表 5-13 给出了 InstanceCache 接口。

列表 5-13 org.jboss.ejb.InstanceCache 接口

```
public interface InstanceCache extends ContainerPlugin  
{  
    /**  
     * Gets a bean instance from this cache given the identity.  
     * This method may involve activation if the instance is not  
     * in the cache.  
     * Implementation should have O(1) complexity.  
     * This method is never called for stateless session beans.  
     *  
     * @param id the primary key of the bean  
     * @return the EnterpriseContext related to the given id  
     * @exception RemoteException in case of illegal calls  
     * (concurrent / reentrant), NoSuchObjectException if  
     * the bean cannot be found.  
     * @see #release  
     */  
    public EnterpriseContext get(Object id)  
        throws RemoteException, NoSuchObjectException;  
    /**  
     * Inserts an active bean instance after creation or activation.  
     * Implementation should guarantee proper locking and O(1) complexity.  
     *  
     * @param ctx the EnterpriseContext to insert in the cache  
     * @see #remove  
     */  
}
```



```
public void insert(EnterpriseContext ctx);

/**
 * Releases the given bean instance from this cache.
 * This method may passivate the bean to get it out of the cache.
 * Implementation should return almost immediately leaving the
 * passivation to be executed by another thread.
 *
 * @param ctx the EnterpriseContext to release
 * @see #get
 */
public void release(EnterpriseContext ctx);

/** Removes a bean instance from this cache given the identity.
 * Implementation should have O(1) complexity and guarantee
 * proper locking.
 *
 * @param id the primary key of the bean
 * @see #insert
 */
public void remove(Object id);

/** Checks whether an instance corresponding to a particular
 * id is active
 *
 * @param id the primary key of the bean
 * @see #insert
 */
public boolean isActive(Object id);
}
```

InstanceCache 除了管理激活的实例列表外，它还负责激活和挂起实例。如果客户请求特定标识的实例，而且它当前并不处于激活状态，InstanceCache 将使用 InstancePool 获取可用的实例，然后使用持久化管理器激活该实例。类似地，如果 InstanceCache 打算挂起某激活的实例，它必须调用持久化管理器挂起它，然后将该实例释放给 InstancePool。

5. org.jboss.ejb.EntityPersistenceManager

EntityPersistenceManager 负责实体 Bean 的持久化。它需要负责如下一些具体任务。

- 在存储源中创建 EJB 实例
- 装载给定主键的状态信息到 EJB 实例中
- 存储给定 EJB 实例的状态
- 从存储源中删除 EJB 实例
- 激活 EJB 实例的状态
- 挂起 EJB 实例的状态

列表 5-14 给出了 EntityPersistenceManager 接口。

列表 5-14 org.jboss.ejb.EntityPersistenceManager 接口

```
public interface EntityPersistenceManager extends ContainerPlugin
{
    /**
     * Returns a new instance of the bean class or a subclass of the bean class.
     *
     * @return the new instance
     */
    Object createBeanClassInstance() throws Exception;

    /**
     * This method is called whenever an entity is to be created. The
     * persistence manager is responsible for calling the ejbCreate method
     * on the instance and to handle the results properly wrt the persistent
     * store.
     *
     * @param m the create method in the home interface that was
     * called
     * @param args any create parameters
     * @param instance the instance being used for this create call
     */
    void createEntity(Method m,
        Object[] args,
        EntityEnterpriseContext instance)
        throws Exception;

    /**
     * This method is called whenever an entity is to be created. The
     * persistence manager is responsible for calling the ejbPostCreate method
     * on the instance and to handle the results properly wrt the persistent
     * store.
     *
     * @param m the create method in the home interface that was
     * called
     * @param args any create parameters
     * @param instance the instance being used for this create call
     */
    void postCreateEntity(Method m,
        Object[] args,
        EntityEnterpriseContext instance)
        throws Exception;

    /**
     * This method is called when single entities are to be found. The
```

```

* persistence manager must find out whether the wanted instance is
* available in the persistence store, and if so it shall use the
* ContainerInvoker plugin to create an EJBObject to the instance, which
* is to be returned as result.
*
* @param finderMethod the find method in the home interface that was
* called
* @param args any finder parameters
* @param instance the instance to use for the finder call
* @return an EJBObject representing the found entity
*/
Object findEntity(Method finderMethod,
    Object[] args,
    EntityEnterpriseContext instance)
    throws Exception;

/**
* This method is called when collections of entities are to be found. The
* persistence manager must find out whether the wanted instances are
* available in the persistence store, and if so it shall use the
* ContainerInvoker plugin to create EJBObjects to the instances, which are
* to be returned as result.
*
* @param finderMethod the find method in the home interface that was
* called
* @param args any finder parameters
* @param instance the instance to use for the finder call
* @return an EJBObject collection representing the found
* entities
*/
Collection findEntities(Method finderMethod,
    Object[] args,
    EntityEnterpriseContext instance)
    throws Exception;

/**
* This method is called when an entity shall be activated. The persistence
* manager must call the ejbActivate method on the instance.
*
* @param instance the instance to use for the activation
*
* @throws RemoteException thrown if some system exception occurs
*/
void activateEntity(EntityEnterpriseContext instance)
    throws RemoteException;

```

```
/**
 * This method is called whenever an entity shall be load from the
 * underlying storage. The persistence manager must load the state from
 * the underlying storage and then call ejbLoad on the supplied instance.
 *
 * @param instance the instance to synchronize
 *
 * @throws RemoteException thrown if some system exception occurs
 */
void loadEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/**
 * This method is used to determine if an entity should be stored.
 *
 * @param instance the instance to check
 * @return true, if the entity has been modified
 * @throws Exception thrown if some system exception occurs
 */
boolean isModified(EntityEnterpriseContext instance) throws Exception;

/**
 * This method is called whenever an entity shall be stored to the
 * underlying storage. The persistence manager must call ejbStore on the
 * supplied instance and then store the state to the underlying storage.
 *
 * @param instance the instance to synchronize
 *
 * @throws RemoteException thrown if some system exception occurs
 */
void storeEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/**
 * This method is called when an entity shall be passivate. The persistence
 * manager must call the ejbPassivate method on the instance.
 *
 * @param instance the instance to passivate
 *
 * @throws RemoteException thrown if some system exception occurs
 */
void passivateEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/**
 * This method is called when an entity shall be removed from the
```



```

* underlying storage. The persistence manager must call ejbRemove on the
* instance and then remove its state from the underlying storage.
*
* @param instance the instance to remove
*
* @throws RemoteException thrown if some system exception occurs
* @throws RemoveException thrown if the instance could not be removed
*/
void removeEntity(EntityEnterpriseContext instance)
    throws RemoteException, RemoveException;
}

```

对于 EJB 2.0 规范, JBoss 支持两种实体 Bean 持久化语义: 容器管理持久化 (Container Managed Persistence, CMP) 和 Bean 管理持久化 (Bean Managed Persistence, BMP)。CMP 实现使用 `org.jboss.ejb.EntityPersistenceStore` 接口。默认情况下, 将 `org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager` 指定为 CMP2 持久化引擎的操作入口。列表 5-15 给出了 `EntityPersistenceStore` 接口。

列表 5-15 `org.jboss.ejb.EntityPersistenceStore` 接口

```

public interface EntityPersistenceStore extends ContainerPlugin
{
    /**
     * Returns a new instance of the bean class or a subclass of the bean class.
     *
     * @return the new instance
     *
     * @throws Exception
     */
    Object createBeanClassInstance() throws Exception;

    /**
     * Initializes the instance context.
     *
     * <p>This method is called before createEntity, and should
     * reset the value of all cmpFields to 0 or null.
     *
     * @param ctx
     *
     * @throws RemoteException
     */
    void initEntity(EntityEnterpriseContext ctx);

    /**
     * This method is called whenever an entity is to be created.
     * The persistence manager is responsible for handling the results properly
     * wrt the persistent store.
     */
}

```

```
*
* @param m the create method in the home interface that was
* called
* @param args any create parameters
* @param instance the instance being used for this create call
* @return The primary key computed by CMP PM or null for BMP
*
* @throws Exception
*/
Object createEntity(Method m,
    Object[] args,
    EntityEnterpriseContext instance)
    throws Exception;

/**
* This method is called when single entities are to be found. The
* persistence manager must find out whether the wanted instance is
* available in the persistence store, if so it returns the primary key of
* the object.
*
* @param finderMethod the find method in the home interface that was
* called
* @param args any finder parameters
* @param instance the instance to use for the finder call
* @return a primary key representing the found entity
*
* @throws RemoteException thrown if some system exception occurs
* @throws FinderException thrown if some heuristic problem occurs
*/
Object findEntity(Method finderMethod,
    Object[] args,
    EntityEnterpriseContext instance)
    throws Exception;

/**
* This method is called when collections of entities are to be found. The
* persistence manager must find out whether the wanted instances are
* available in the persistence store, and if so it must return a
* collection of primaryKeys.
*
* @param finderMethod the find method in the home interface that was
* called
* @param args any finder parameters
* @param instance the instance to use for the finder call
* @return an primary key collection representing the found
* entities
```

```

*
* @throws RemoteException thrown if some system exception occurs
* @throws FinderException thrown if some heuristic problem occurs
*/
Collection findEntities(Method finderMethod,
    Object[] args,
    EntityEnterpriseContext instance)
    throws Exception;

/**
* This method is called when an entity shall be activated.
*
* <p>With the PersistenceManager factorization most EJB calls should not
* exists However this calls permits us to introduce optimizations in
* the persistence store. Particularly the context has a
* "PersistenceContext" that a PersistenceStore can use (JAWS does for
* smart updates) and this is as good a callback as any other to set it
* up.
* @param instance the instance to use for the activation
*
* @throws RemoteException thrown if some system exception occurs
*/
void activateEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/**
* This method is called whenever an entity shall be load from the
* underlying storage. The persistence manager must load the state from
* the underlying storage and then call ejbLoad on the supplied instance.
*
* @param instance the instance to synchronize
*
* @throws RemoteException thrown if some system exception occurs
*/
void loadEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/**
* This method is used to determine if an entity should be stored.
*
* @param instance the instance to check
* @return true, if the entity has been modified
* @throws Exception thrown if some system exception occurs
*/
boolean isModified(EntityEnterpriseContext instance) throws Exception;

```

```
/**
 * This method is called whenever an entity shall be stored to the
 * underlying storage. The persistence manager must call ejbStore on the
 * supplied instance and then store the state to the underlying storage.
 *
 * @param instance the instance to synchronize
 *
 * @throws RemoteException thrown if some system exception occurs
 */
void storeEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/**
 * This method is called when an entity shall be passivate. The persistence
 * manager must call the ejbPassivate method on the instance.
 *
 * <p>See the activate discussion for the reason for exposing EJB callback
 * calls to the store.
 *
 * @param instance the instance to passivate
 *
 * @throws RemoteException thrown if some system exception occurs
 */
void passivateEntity(EntityEnterpriseContext instance)
    throws RemoteException;

/**
 * This method is called when an entity shall be removed from the
 * underlying storage. The persistence manager must call ejbRemove on the
 * instance and then remove its state from the underlying storage.
 *
 * @param instance the instance to remove
 *
 * @throws RemoteException thrown if some system exception occurs
 * @throws RemoveException thrown if the instance could not be removed
 */
void removeEntity(EntityEnterpriseContext instance)
    throws RemoteException, RemoveException;
}
```

org.jboss.ejb.plugins.BMPPersistenceManager 是 EntityPersistenceManager 接口的默认 BMP 实现。由于 BMP 所有的持久化逻辑都是由其本身完成，所以 BMP 持久化管理器相当简单。BMP 持久化管理器惟一的职责是完成容器回调。

6. org.jboss.ejb.StatefulSessionPersistenceManager

StatefulSessionPersistenceManager 负责有状态 SessionBean 的持久化。它负责的具体任

务有如下一些：

- 在存储源中创建有状态会话。
- 激活存储源中的有状态会话。
- 将有状态会话挂起到存储源中。
- 从存储源中删除有状态会话。

列表 5-16 给出了 `StatefulSessionPersistenceManager` 接口。

列表 5-16 `org.jboss.ejb.StatefulSessionPersistenceManager` 接口

```
public interface StatefulSessionPersistenceManager extends
    ContainerPlugin
{
    public void createSession(Method m, Object[] args,
        StatefulSessionEnterpriseContext ctx)
        throws Exception;

    public void activateSession(StatefulSessionEnterpriseContext ctx)
        throws RemoteException;

    public void passivateSession(StatefulSessionEnterpriseContext ctx)
        throws RemoteException;

    public void removeSession(StatefulSessionEnterpriseContext ctx)
        throws RemoteException, RemoveException;

    public void removePassivated(Object key);
}
```

`org.jboss.ejb.plugins.StatefulSessionFilePersistenceManager` 类是接口，即 `StatefulSessionPersistenceManager` 的默认实现。从名字上可以看出，该实现使用文件系统持久化有状态会话 Bean。进一步而言，持久化管理器将企业 Bean 序列化在平面文件中，其文件名由会话 Bean 名和会话 id 构成、扩展名为 `.ser`。持久化管理器在激活企业 Bean 以获得企业 Bean 状态和在挂起企业 Bean 以保存企业 Bean 状态期间，都需要使用企业 Bean 的 `.ser` 文件。

5.4 实体 Bean 锁和死锁检测

本节内容将讨论 JBoss 3.x 中实体 Bean 锁的含义、如何访问和锁住实体 Bean。同时也将讨论开发者使用实体 Bean 过程中可能碰到的问题及如何解决它们。然后，本文很正规地给出死锁的定义和检测。最后，本文给出了如何根据实体 Bean 锁来对目标系统进行调优。

5.4.1 JBoss 为什么需要锁

锁，是为保护数据集成性而存在的。有时，在特定时间内只允许有一个用户能够更新

关键数据。有时，需要以串行方式访问系统中的敏感对象，以避免并发读写操作而引起的数据瘫痪。通常，数据库借助于事务范围提供了这种功能，比如锁表、锁行服务。

实体 Bean 为关系型数据提供了优良的面向对象接口。而且，通过缓存和延迟更新数据库内容，直到确实需要立即更新数据。最终，便能够降低数据库负荷以改善性能、大大提高了数据库的效率。但是，引入缓存后，数据集成性是一个大问题。因此，需要为实体 Bean 提供某种应用服务器级锁，从而能够和传统的数据库一起使用事务隔离属性。

5.4.2 实体 Bean 的生命周期

JBoss 提供的默认配置只允许同时在内存中有某实体 Bean 的一个激活实例。当然，通过缓存配置和 commit-option 类型能够更改其行为。对于各个 commit-option 而言，企业 Bean 实例的生命周期是不同的。

- 对于 commit-option, “A”: 实例能够在事务之间受到缓存和使用。
- 对于 commit-option, “B”: 实例能够在事务之间受到缓存和使用。但是，事务结束时会将该实例标记为 “dirty”。其意味着在启动新事务时，必须调用 ejbLoad 方法。
- 对于 commit-option, “C”: 在事务结束时，会将实例标记为 “dirty”，并从缓存中释放掉，然后挂起它。
- 对于 commit-option, “D”: 在缓存中存在后台刷新线程，能够定期调用陈旧 (stale) 企业 Bean 的 ejbLoad 方法。其他方面，同选项 “A”。

其中，标记为挂起的含义为将该企业 Bean 放置到队列中。每个实体 Bean 容器都存在 Passivation 线程，即定期将企业 Bean 放置在队列中。如果应用请求需要访问同一主键实体 Bean，则将该企业 Bean 从挂起队列中拿出以供客户使用。

一旦有异常抛出或出现事务回滚，实体 Bean 实例将完全从缓存中抛出，甚至不会再次放入到挂起队列中，实例池也不会再次使用它。除挂起队列外，再无实体 Bean 池。由于过多的存储内容会导致太多的问题和 Bug，因此没有引入实体 Bean 池。

5.4.3 默认锁行为

实体 Bean 锁已经完全从实体 Bean 实例中独立出来。用于锁的逻辑完全存放和管理在单独的锁对象中。因此，能够更好地实现实体 Bean 生命周期，在后文中将有深入阐述。

因为同时只允许存在一个实体 Bean 实例处于激活状态，所以 JBoss 使用了两种类型锁，以确保数据集成性和遵循 EJB 规范。

1. 方法锁

方法锁能够保证同时只有单个线程访问特定实体 Bean，这也是 EJB 规范定义的。但是，通过部署描述符将企业 Bean 标记为可重入 (reentrant)，便能覆盖这种单线程特性。

2. 事务锁

事务锁能够确保同时只有单个事务访问特定实体 Bean，从而在应用服务器级保证事

事务的 ACID 属性。既然在默认情况下同时只有单个事务访问特定实体 Bean，则 JBoss 必须保护该实例，以防止脏读和脏写。因此，直到事务完成，默认的实体 Bean 锁行为将一直锁住实体 Bean。其含义为，如果调用事务中实体 Bean 的任何方法，除非持有该企业 Bean 的事务成功提交或回滚，否则其他事务不能访问到该企业 Bean。

5.4.4 插入式拦截器和锁策略

本文接下来看看，在 standardjboss.xml 描述符定义的容器配置中给出的实体 Bean 生命周期和行为。列表 5-17 给出了“Standard CMP 2.x EntityBean”配置中的 container-interceptors 定义。

列表 5-17 “Standard CMP 2.x EntityBean”拦截器定义

```
<container-configuration>
  <container-name>Standard CMP 2.x EntityBean</container-name>
  ...
  <container-interceptors>
    <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.TxInterceptorCMP</interceptor>
    <interceptor>org.jboss.ejb.plugins.MetricsInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.EntityCreationInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.EntityLockInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.EntityInstanceInterceptor</interceptor>
    <interceptor>org.jboss.resource.connectionmanager.CachedConnectionInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.EntitySynchronizationInterceptor</interceptor>
    <interceptor>org.jboss.ejb.plugins.cmp.jdbc.JDBCRelationInterceptor</interceptor>
  </container-interceptors>
```

上述给出的拦截器定义了实体 Bean 的大部分行为。如下给出了与本节相关拦截器的解释。

- **EntityLockInterceptor**。该拦截器的作用是在处理调用之前安排好所需要的锁。该拦截器非常轻量，其将所有的锁行为都委派给插入式锁策略。
- **EntityInstanceInterceptor**。该拦截器的工作是在缓存中寻找实体 Bean 或创建新的实体 Bean。该拦截器也能够确保，内存中同时只有一个激活实例。
- **EntitySynchronizationInterceptor**。该拦截器的职责是同步底端存储源和缓存状态。通过使用 EJB 规范中定义的 ejbLoad 和 ejbStore 语义能实现上述目的。如果存在事务，则事务声明将触发该拦截器。它通过 JTA 接口注册了底层事务监控器，以作为回调。如果没有事务，则存储状态信息到调用返回值中。该拦截器处理规范给出的同步策略 A、B、C，以及 JBoss 扩展的 commit-option 值“D”。

5.4.5 死锁

本节讨论寻找和解决死锁问题。具体将阐述：何为死锁、如何检测应用中的死锁及如何消除死锁。当两个或多个线程锁住共享资源时，会出现死锁。比如，图 5-11 阐述了一个简单的死锁场景。其中，线程 1 具有 Bean A 的锁、线程 2 具有 Bean B 的锁。后来，线程 1 试图锁住 Bean B，但阻塞了，因为线程 2 已经锁住 Bean B 了。反之亦然，线程 2 试图锁住 Bean A，但阻塞了，因为线程 1 已经锁住 Bean A 了。此时此刻，两个线程形成了死锁，它们所访问的资源已经被对方锁住了。

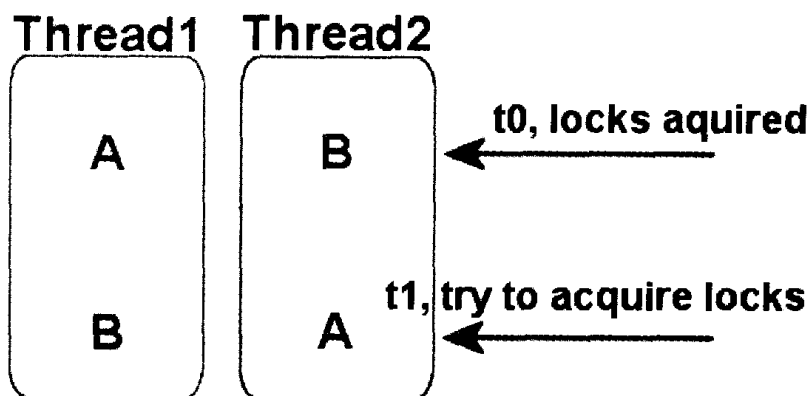


图 5-11 死锁定义实例

JBoss 的默认锁策略是，当调用出现在事务上下文中时，如果事务没有结束，则一直锁住实体 Bean。据此，如果应用中有访问多个实体 Bean 的长事务，或者没有注意实体 Bean 的访问顺序，则容器触发死锁。为避免死锁问题，可以使用不同的技术和高级配置。本节后续内容将有讨论。

死锁检测

幸运的是，自从 JBoss 2.4.5 以来，在基代码中引入了死锁检测算法。JBoss 持有等待事务的全局内部图和哪些事务处于阻塞状态。只要线程不能够获得实体 Bean 锁时，它便能计算出那个事务当前持有该实体 Bean 锁，然后将其本身添加到阻塞事务图中。表 5-1 给出了事务图的实例。

表 5-1 阻塞事务表实例

阻塞 Tx	持有所需锁的 Tx
Tx1	Tx2
Tx2	Tx4
Tx3	Tx1

在线程真正阻塞前，它会检测是否出现了死锁问题，即通过仔细研究阻塞事务图的方式。在研究过程中，它会记录哪些事务处于阻塞状态。如果某阻塞节点出现了多次，则存在死锁，从而触发该事务释放其持有的所有锁。可以通过 BeanLockSupport 的 deadlock Detection 方法获得死锁检测算法。列表 5-18 给出了相应的代码。

列表 5-18 org.jboss.ejb.plugins.lock.BeanLockSupport 中的 deadlockDetection 方法

```
// This following is for deadlock detection
protected static HashMap waiting = new HashMap();

public void deadlockDetection(Transaction miTx) throws Exception
{
    HashSet set = new HashSet();
    set.add(miTx);

    Object checkTx = this.tx;
    synchronized(waiting)
    {
        while (checkTx != null)
        {
            Object waitingFor = waiting.get(checkTx);
            if (waitingFor != null)
            {
                if (set.contains(waitingFor))
                {
                    log.error("Application deadlock detected: " + miTx + " has deadlock conditions");
                    throw new ApplicationDeadlockException("application deadlock detected");
                }
                set.add(waitingFor);
            }
            checkTx = waitingFor;
        }
    }
}
```

(1) 捕捉 ApplicationDeadlockException

既然 JBoss 能够检测到应用死锁，则如果由于 ApplicationDeadlockException 而导致调用失败时，可以在应用中再次启动该事务。但不幸的是，该异常深深地嵌入在 RemoteException 中，因此需要在 catch 块中搜索它，举例如下。

```
try
{
    ...
}
catch (RemoteException ex)
{
    Throwable cause = null;
    RemoteException rex = ex;
    while (rex.detail != null)
    {
        cause = rex.detail;
```



```
if (cause instanceof ApplicationDeadlockException)
{
    // ... We have deadlock, force a retry of the transaction.
    break;
}
if (cause instanceof RemoteException)
{
    dex = (RemoteException)cause;
}
}
}
```

（2）浏览锁信息

JBoss 3.0.4 添加的新 MBean 服务，即 `org.jboss.monitor.EntityLockMonitor`，允许客户浏览基本的锁统计信息，也包括事务锁表状态信息的打印。为使用该监控器，开发者需要使用 `conf/jboss-service.xml` 中的如下配置。

```
<mbean code="org.jboss.monitor.EntityLockMonitor"
       name="jboss.monitor:name=EntityLockMonitor"/>
```

`EntityLockMonitor` 不存在可配置属性。但是，它存在如下只读属性。

- **MedianWaitTime**：线程为获得锁而等待的所有时间的中数（median value）。
- **AverageContentenders**：所有等待锁的线程数量与 `contentions` 数量的比例。
- **TotalContentions**：为获得事务锁，而等待的全部线程数量。具体发生在如下场合，即当线程试图获得其他事务关联的锁时。
- **MaxContentenders**：为获得事务锁，而等待的最大线程数量。

另外，还有如下操作：

- **clearMonitor**：该操作通过将所有计数器设成 0，以复位锁监控器。
- **printLockMonitor**：该操作打印出所有 EJB 锁表，即企业 Bean 的 `ejb-name`。另外，还包括等待锁的总时间、等待锁花费的次数及那些因等待锁而超时的事务数量。

5.4.6 高级配置和调优

默认的实体 Bean 锁行为可能引起死锁。既然在访问实体 Bean 时将该 Bean 锁住在事务中，这对应用而言也引入了不可忽视的性能（吞吐量）问题。本节内容讨论如何通过各种技术和配置来优化性能，并减少发生死锁的可能性。

1. 短期事务

尽可能将事务实现为短期的、细粒度的。事务越短，越不容易产生并发访问冲突，从而使应用的吞吐量更高。

2. 顺序访问

顺序访问实体 Bean 有助于减少死锁发生的可能性，即在系统中总是以相同的顺序访

问实体 Bean。大部分情况下，由于用户应用过于复杂而不能使用这种方法，因此需要更高级的配置。

3. 只读实体 Bean

开发者可以将实体 Bean 标记为只读。当某实体 Bean 被标记为只读时，即意味着它再也不会参与事务了。所以，此时一定不会发生事务锁。在某种场合，将只读选项和 commit-option “D” 一起使用非常有效，比如外部数据源偶尔会更新只读实体 Bean 的数据时。

开发者使用 jboss.xml 部署描述符中的 <read-only> 标签能够实现，对只读实体 Bean 进行标记，如列表 5-19 所示。

列表 5-19 使用 jboss.xml 标记只读实体 Bean

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>MyEntityBean</ejb-name>
      <jndi-name>MyEntityHomeRemote</jndi-name>
      <read-only>True</read-only>
    </entity>
  </enterprise-beans>
</jboss>
```

4. 显式地定义只读方法

在阅读和理解实体 Bean 的默认锁行为后，开发者可能存在这样的疑惑，“如果没有修改数据，为什么要锁住实体 Bean?”。JBoss 允许开发者定义实体 Bean 的方法级只读。因此，应用只调用到这些只读方法时，该实体 Bean 并不会发生锁行为。使用 jboss.xml 部署描述符能够定义只读方法。其中，允许使用通配符表示方法名。列表 5-20 给出了实例，其将以 “get” 开头的所有方法、并将 anotherReadOnlyMethod 标记为只读。

列表 5-20 将实体 Bean 方法定义为只读

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>nextgen.EnterpriseEntity</ejb-name>
      <jndi-name>nextgen.EnterpriseEntity</jndi-name>
      <method-attributes>
        <method>
          <method-name>get*</method-name>
          <read-only>true</read-only>
        </method>
        <method>
          <method-name>anotherReadOnlyMethod</method-name>
          <read-only>true</read-only>
        </method>
      </method-attributes>
    </entity>
  </enterprise-beans>
</jboss>
```

```
</method>
</method-attributes>
</entity>
</enterprise-beans>
</jboss>
```

5. Instance Per Transaction 策略

Instance Per Transaction 策略属于高级配置,它能够完全消除由 JBoss 默认锁策略引起的死锁和吞吐量问题。实体 Bean 的默认锁策略是,只允许存在企业 Bean 的单个激活实例。Instance Per Transaction 策略打破这种限制,即每次事务开始时分配新的企业 Bean 实例,在事务结束时丢弃该实例。由于每个事务都存在企业 Bean 的单独拷贝,因此不需要基于锁的事务。

该主意听起来不错,但确实还存在一些问题。首先,其事务隔离行为等价于 READ_COMMITTED。从而,能出现重复读操作。换句话说,该事务拥有了陈旧实体 Bean 的拷贝。其次,该配置选项目前还是需要 commit-option “B” 或者 “C”,从而在性能上大打折扣,因为在启动事务时需要执行 ejbLoad。但如果应用本身要求使用 commit-option “B” 或者 “C”,则使用 Instance Per Transaction 策略是好办法。JBoss 开发者当前也在探索使用 commit-option “A” 的方式,以后则能够使用缓存功能了。

JBoss 服务器中存在的一个容器配置,即 standardjboss.xml,就实现了这种锁策略,对应的容器配置名为“Instance Per Transaction CMP 2.x EntityBean”和“Instance Per Transaction BMP EntityBean”。如果需要使用这些配置,则需要开发者在 jboss.xml 部署描述符中引用待使用的容器配置名,如列表 5-21 所示。

列表 5-21 使用 “Instance Per Transaction” 策略实例 (JBoss 3.0.1 以上)

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>MyCMP2Bean</ejb-name>
      <jndi-name>MyCMP2</jndi-name>
      <configuration-name>Instance Per Transaction CMP 2.x EntityBean</configuration-name>
    </entity>
    <entity>
      <ejb-name>MyBMPBean</ejb-name>
      <jndi-name>MyBMP</jndi-name>
      <configuration-name>Instance Per Transaction BMP EntityBean</configuration-name>
    </entity>
  </enterprise-beans>
</jboss>
```

如果开发者正在使用 JBoss 3.0.0,则需要在 jboss.xml 描述符中设置自身的配置。在列表 5-22 中,高亮显示的代码就是新加的配置信息。

列表 5-22 “Instance Per Transaction” 配置

```

<jboss>
  <container-configurations>
    <container-configuration>
      <container-name>Instance Per Transaction CMP 2.x EntityBean</container-name>
      <call-logging>>false</call-logging>
      <container-invoker>org.jboss.proxy.ejb.ProxyFactory</container-invoker>
      <container-interceptors>
        <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>
        <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>
        <interceptor>org.jboss.ejb.plugins.TxInterceptorCMT</interceptor>
        <interceptor metricsEnabled = "true">org.jboss.ejb.plugins.MetricsInterceptor</interceptor>
        <interceptor>org.jboss.ejb.plugins.EntityCreationInterceptor</interceptor>
        <interceptor>org.jboss.ejb.plugins.EntityLockInterceptor</interceptor>
        <interceptor>org.jboss.ejb.plugins.EntityMultiInstanceInterceptor</interceptor>
        <interceptor>org.jboss.resource.connectionmanager.CachedConnectionInterceptor</interceptor>
        <interceptor>org.jboss.ejb.plugins.EntityMultiInstanceSynchronizationInterceptor
        </interceptor>
        <interceptor>org.jboss.ejb.plugins.cmp.jdbc.JDBCRelationInterceptor</interceptor>
      </container-interceptors>
      <client-interceptors>
        <home>
          <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
          <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
          <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
          <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
        </home>
        <bean>
          <interceptor>org.jboss.proxy.ejb.EntityInterceptor</interceptor>
          <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
          <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
          <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
        </bean>
        <list-entity>
          <interceptor>org.jboss.proxy.ejb.ListEntityInterceptor</interceptor>
          <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
          <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
          <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
        </list-entity>
      </client-interceptors>
      <instance-pool>org.jboss.ejb.plugins.EntityInstancePool</instance-pool>
      <instance-cache>org.jboss.ejb.plugins.EntityInstanceCache</instance-cache>
      <persistence-manager>org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager</persistence-manager>
      <transaction-manager>org.jboss.tm.TxManager</transaction-manager>
      <locking-policy>org.jboss.ejb.plugins.lock.MethodOnlyEJBLock</locking-policy>
    </container-configuration>
  </container-configurations>
</jboss>

```

```
<container-cache-conf>
  <cache-policy>org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy</cache-policy>
  <cache-policy-conf>
    <min-capacity>50</min-capacity>
    <max-capacity>1000000</max-capacity>
    <overager-period>300</overager-period>
    <max-bean-age>600</max-bean-age>
    <resizer-period>400</resizer-period>
    <max-cache-miss-period>60</max-cache-miss-period>
    <min-cache-miss-period>1</min-cache-miss-period>
    <cache-load-factor>0.75</cache-load-factor>
  </cache-policy-conf>
</container-cache-conf>
<container-pool-conf>
  <MaximumSize>100</MaximumSize>
</container-pool-conf>
<commit-option>B</commit-option>
</container-configuration>
```

5.4.7 在群集中运行

当前, JBoss 并没有为运行于群集中的实体 Bean 提供分布式锁的支持。该项功能已经委派给了数据库, 应用开发者必须支持它。对于群集实体 Bean 而言, 建议开发者在使用 commit-option “B” 或 “C” 时, 也一起使用行锁机制。对于 CMP, 存在配置选项 <row-locking>。当实体 Bean 从数据库装入时, <row-locking>值可以是 “SELECT ... FOR UPDATE”。对于 commit-option “B” 或 “C”, 其实现的事务锁能够用于群集环境中。对于 BMP 而言, 开发者必须在 BMP 的 ejbLoad 方法中显式地实现 “SELECT ... FOR UPDATE” 调用。

5.4.8 修理故障

本节内容将阐述一些常见锁问题及其解决办法。

1. 锁行为失效

在 JBoss 用户邮件列表中有许多邮件提到, 有时候锁功能失效。他们需要并发访问实体 Bean, 因此使用了脏读。具体解决办法如下。

- 如果从 JBoss 2.4.0 或以下版本升级, 并存在自定义 <container-configurations>, 则开发者必须确保已更新了这些配置, 因为这些版本的 JBoss 服务器没有提供 EntityLockInterceptor。
- 开发者务必正确地实现, 自定义或复杂主键类的 equals 和 hashCode 方法。
- 开发者务必正确地实现, 自定义或复杂主键类的序列化。一种常见的错误是, 还没有对主键进行解包时, 就执行了成员变量的初始化工作。

2. IllegalStateException

有时，开发者获得：`java.lang.IllegalStateException: removing bean lock and it has tx set!`

其含义为，开发者没有为自定义或复杂主键类正确地实现 `equals` 或 `hashCode` 方法，或者没有正确地实现主键类的序列化工作。

3. 悬挂和事务超时

JBoss 很久没有解决的一个 Bug，即事务超时。该事务仅仅打着旗号说已经回滚，而实际上根本没有回滚。开发者可以考虑通过调用线程解决。但如果调用线程一直悬挂（hang），比如由于实体 Bean 锁没有被释放，就会引起这样的严重问题。因此，这并不是解决问题的好办法。真正解决问题的办法是，开发者需要避免在事务中触发一直悬挂的操作。其中，最大的禁忌在于，不要在事务中进行跨越互联网连接或运行 Web 爬行器（web-crawler）。

第6章 JBoss 之消息——JMS

配置和架构

JMS API 代表 Java 消息服务应用编程接口，应用程序使用它为其他应用发送异步“业务特征”的消息。在 JMS 中，消息并没有直接发送到其他应用中，而是发送到目的地（destination），即“queue”或者“topic”。发送消息的应用程序并不需要担心消息接收应用是否已启动并且在运行。相反，消息接收应用也不用担心发送方的状态，发送者和接收者仅仅和目的地交互。

JMS API 是为 JMS 供应商提供的标准接口，有时称为面向消息中间件（Message Oriented Middleware, MOM）系统。JBoss 实现的、兼容于 JMS 1.02b 规范的 JMS 供应商称为 JBoss 消息，或 JBossMQ。当在 JBoss 中使用 JMS API 时，开发者能够透明地使用 JBoss 消息引擎。JBoss 消息完整地实现了 JMS 规范。因此，最好的 JBoss 消息用户指南是 JMS 规范！更多 JMS API 信息，请开发者访问 JMS 教程（http://java.sun.com/products/jms/tutorial/1_3_1-fcs/doc/copyright.html）或 JMS 下载和规范（<http://java.sun.com/products/jms/docs.html>）。

本章重点关注几方面的内容：在 JBoss 中使用 JMS 和消息驱动 Bean、JBoss 消息配置和 MBean 服务。

6.1 JMS 实例

本节内容讨论，为使用 JBoss JMS 实现所需的基础知识。JMS 将访问 JMS 连接工厂和目的地细节留给了供应商去实现。学会使用 JBoss 消息层，只需要开发者掌握如下知识。

- `javax.jms.QueueConnectionFactory` 和 `javax.jms.TopicConnectionFactory` 的位置。在 JBoss 中，这两个连接工厂实现都位于 JNDI 名“ConnectionFactory”下。
- 如何查找 JMS 目的地（`javax.jms.Queue` 和 `javax.jms.Topic`）。当本文讨论消息 MBean 时，开发者将看到如何借助 MBean 配置目的地。JBoss 定义了若干个默认 queue，其 JNDI 名分别为：“queue/testQueue”、“queue/ex”、“queue/A”、“queue/B”、“queue/C”及“queue/D”。默认的 topic 位于 JNDI 名：“topic/testTopic”、“topic/securedTopic”及“topic/testDurableTopic”。
- JBoss 消息 jar 文件，具体包括如下内容：`concurrent.jar`、`jbossmq-client.jar`、`jboss-common-client.jar`、`jboss-system-client.jar`、`jnp-client.jar`、`log4j.jar` 及 `jnet.jar`（用于 JDK 1.3）。

接下来的内容，本章将给出不同的 JMS 消息模型和消息驱动 Bean。其源代码位于 `src/main/org/jboss/chap6` 目录中。

6.1.1 点对点实例

本文以点对点（point-to-point, P2P）实例为切入点。在 P2P 模型中，发送者分发消息到 queue 中，单个接收者将消息从 queue 中移出。接收者并不需要在消息发送的同时监听该 queue。列表 6-1 给出了某个完整的 P2P 实例，其将 javax.jms.TextMessage 发送到 queue，即 “queue/testQueue”，并以异步方式接收了该 queue 中的消息。

列表 6-1 P2P JMS 客户端实例

```
package org.jboss.chap6.ex1;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueReceiver;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import edu.oswego.cs.dl.util.concurrent.CountDown;

/** A complete JMS client example program that sends a
    TextMessage to a Queue and asynchronously receives the
    message from the same Queue.

    @author Scott.Stark@jboss.org
    @version $Revision:$
    */
public class SendRecvClient
{
    static CountDown done = new CountDown(1);
    QueueConnection conn;
    QueueSession session;
    Queue que;

    public static class ExListener implements MessageListener
    {
        public void onMessage(Message msg)
        {
            done.release();
            TextMessage tm = (TextMessage) msg;
        }
    }
}
```

```
try
{
    System.out.println("onMessage, recv text="
        + tm.getText());
}
catch(Throwable t)
{
    t.printStackTrace();
}
}

}

public void setupPTP()
    throws JMSEException, NamingException
{
    InitialContext iniCtx = new InitialContext();
    Object tmp = iniCtx.lookup("ConnectionFactory");
    QueueConnectionFactory qcf = (QueueConnectionFactory) tmp;
    conn = qcf.createQueueConnection();
    que = (Queue) iniCtx.lookup("queue/testQueue");
    session = conn.createQueueSession(false,
        QueueSession.AUTO_ACKNOWLEDGE);
    conn.start();
}

public void sendRecvAsync(String text)
    throws JMSEException, NamingException
{
    System.out.println("Begin sendRecvAsync");
    // Setup the PTP connection, session
    setupPTP();
    // Set the async listener
    QueueReceiver recv = session.createReceiver(que);
    recv.setMessageListener(new ExListener());
    // Send a text msg
    QueueSender send = session.createSender(que);
    TextMessage tm = session.createTextMessage(text);
    send.send(tm);
    System.out.println("sendRecvAsync, sent text="
        + tm.getText());
    send.close();
    System.out.println("End sendRecvAsync");
}

public void stop() throws JMSEException
{
}
```



```
        conn.stop();
        session.close();
        conn.close();
    }

    public static void main(String args[]) throws Exception
    {
        SendRecvClient client = new SendRecvClient();
        client.sendRecvAsync("A text msg");
        client.done.acquire();
        client.stop();
        System.exit(0);
    }
}
```

客户使用如下命令行。

```
[nr@toki examples]$ ant -Dchap=chap6 -Dex=lp2p run-example
Buildfile: build.xml
...
run-examplelp2p:
[java] [INFO,SendRecvClient] Begin SendRecvClient, now=1083649316059
[java] [INFO,SendRecvClient] Begin sendRecvAsync
[java] [INFO,SendRecvClient] onMessage, recv text=A text msg
[java] [INFO,SendRecvClient] sendRecvAsync, sent text=A text msg
[java] [INFO,SendRecvClient] End sendRecvAsync
[java] [INFO,SendRecvClient] End SendRecvClient

BUILD SUCCESSFUL
Total time: 9 seconds
```

6.1.2 发布/订阅实例

JMS 发布/订阅（publish/subscribe, Pub-Sub）消息模型是一对多的模型。发布者将消息发送到 topic，该 topic 的所有活动订阅者将接收该消息。没有监听该 topic 的非活动订阅者将错过这个已发布的消息。列表 6-2 给出了完整的 JMS 客户代码，其发送 javax.jms.TextMessage 到 topic，并从该 topic 异步接收该消息。

列表 6-2 发布/订阅 JMS 客户实例

```
package org.jboss.chap6.ex1;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Topic;
```

```
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSubscriber;
import javax.jms.TopicSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import EDU.oswego.cs.dl.util.concurrent.CountDown;

/** A complete JMS client example program that sends a
    TextMessage to a Topic and asynchronously receives the
    message from the same Topic.

    @author Scott.Stark@jboss.org
    @version $Revision:$
    */
public class TopicSendRecvClient
{
    static CountDown done = new CountDown(1);
    TopicConnection conn = null;
    TopicSession session = null;
    Topic topic = null;

    public static class ExListener implements MessageListener
    {
        public void onMessage(Message msg)
        {
            done.release();
            TextMessage tm = (TextMessage) msg;
            try
            {
                System.out.println("onMessage, recv text="
                    + tm.getText());
            }
            catch(Throwable t)
            {
                t.printStackTrace();
            }
        }
    }

    public void setupPubSub()
        throws JMSEException, NamingException
    {
```

```
        InitialContext iniCtx = new InitialContext();
        Object tmp = iniCtx.lookup("ConnectionFactory");
        TopicConnectionFactory tcf = (TopicConnectionFactory) tmp;
        conn = tcf.createTopicConnection();
        topic = (Topic) iniCtx.lookup("topic/testTopic");
        session = conn.createTopicSession(false,
            TopicSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }

    public void sendRecvAsync(String text)
        throws JMSEException, NamingException
    {
        System.out.println("Begin sendRecvAsync");
        // Setup the PubSub connection, session
        setupPubSub();
        // Set the async listener

        TopicSubscriber recv = session.createSubscriber(topic);
        recv.setMessageListener(new ExListener());
        // Send a text msg
        TopicPublisher send = session.createPublisher(topic);
        TextMessage tm = session.createTextMessage(text);
        send.publish(tm);
        System.out.println("sendRecvAsync, sent text="
            + tm.getText());
        send.close();
        System.out.println("End sendRecvAsync");
    }

    public void stop() throws JMSEException
    {
        conn.stop();
        session.close();
        conn.close();
    }

    public static void main(String args[]) throws Exception
    {
        System.out.println("Begin TopicSendRecvClient,
            now="+System.currentTimeMillis());
        TopicSendRecvClient client = new TopicSendRecvClient();
        client.sendRecvAsync("A text msg,
            now="+System.currentTimeMillis());
        client.done.acquire();
        client.stop();
    }
}
```

```

        System.out.println("End TopicSendRecvClient");
        System.exit(0);
    }

}

```

客户使用如下命令行。

```

[nr@toki examples]$ ant -Dchap=chap6 -Dex=1ps run-example
Buildfile: build.xml
...
run-example1ps:
    [java] Begin TopicSendRecvClient, now=1083649449640
    [java] Begin sendRecvAsync
    [java] onMessage, recv text=A text msg, now=1083649449642
    [java] sendRecvAsync, sent text=A text msg, now=1083649449642
    [java] End sendRecvAsync
    [java] End TopicSendRecvClient

BUILD SUCCESSFUL
Total time: 3 seconds

```

然后，本文将发布者和订阅者分开到独立的程序中，从而证明当订阅者在监听 `topic` 时，它只能接收消息。列表 6-3 给出了前述发布/订阅客户端的另一版本，即仅仅将消息发布到 `topic`，即“`topic/testTopic`”。列表 6-4 仅仅给出了订阅者。

列表 6-3 JMS 发布者客户端

```

package org.jboss.chap6.ex1;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSubscriber;
import javax.jms.TopicSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/** A JMS client example program that sends a TextMessage to a Topic

@author Scott.Stark@jboss.org
@version $Revision:$
*/

```

```
public class TopicSendClient
{
    TopicConnection conn = null;
    TopicSession session = null;
    Topic topic = null;

    public void setupPubSub()
        throws JMSEException, NamingException
    {
        InitialContext iniCtx = new InitialContext();
        Object tmp = iniCtx.lookup("ConnectionFactory");
        TopicConnectionFactory tcf = (TopicConnectionFactory) tmp;
        conn = tcf.createTopicConnection();
        topic = (Topic) iniCtx.lookup("topic/testTopic");
        session = conn.createTopicSession(false,
            TopicSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }

    public void sendAsync(String text)
        throws JMSEException, NamingException
    {
        System.out.println("Begin sendAsync");
        // Setup the pub/sub connection, session
        setupPubSub();
        // Send a text msg
        TopicPublisher send = session.createPublisher(topic);
        TextMessage tm = session.createTextMessage(text);
        send.publish(tm);
        System.out.println("sendAsync, sent text="
            + tm.getText());
        send.close();
        System.out.println("End sendAsync");
    }

    public void stop() throws JMSEException
    {
        conn.stop();
        session.close();
        conn.close();
    }

    public static void main(String args[]) throws Exception
    {
        System.out.println("Begin TopicSendClient,
            now="+System.currentTimeMillis());
        TopicSendClient client = new TopicSendClient();
    }
}
```



```

        client.sendAsync("A text msg,
            now="+System.currentTimeMillis());
        client.stop();
        System.out.println("End TopicSendClient");
        System.exit(0);
    }
}

```

列表 6-4 JMS 订阅者客户端

```

package org.jboss.chap6.ex1;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSubscriber;
import javax.jms.TopicSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/** A JMS client example program that synchronously receives a
    message a Topic

    @author Scott.Stark@jboss.org
    @version $Revision:$
    */
public class TopicRecvClient
{
    TopicConnection conn = null;
    TopicSession session = null;
    Topic topic = null;

    public void setupPubSub()
        throws JMSEException, NamingException
    {
        InitialContext iniCtx = new InitialContext();
        Object tmp = iniCtx.lookup("ConnectionFactory");
        TopicConnectionFactory tcf = (TopicConnectionFactory) tmp;
        conn = tcf.createTopicConnection();
        topic = (Topic) iniCtx.lookup("topic/testTopic");
        session = conn.createTopicSession(false,

```

```
        TopicSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }

    public void recvSync()
        throws JMSEException, NamingException
    {
        System.out.println("Begin recvSync");
        // Setup the pub/sub connection, session
        setupPubSub();
        // Wait upto 5 seconds for the message
        TopicSubscriber recv = session.createSubscriber(topic);
        Message msg = recv.receive(5000);
        if( msg == null )
            System.out.println("Timed out waiting for msg");
        else
            System.out.println("TopicSubscriber.recv, msgt="+msg);
    }

    public void stop() throws JMSEException
    {
        conn.stop();
        session.close();
        conn.close();
    }

    public static void main(String args[]) throws Exception
    {
        System.out.println("Begin TopicRecvClient,
            now="+System.currentTimeMillis());
        TopicRecvClient client = new TopicRecvClient();
        client.recvSync();
        client.stop();
        System.out.println("End TopicRecvClient");
        System.exit(0);
    }
}
```

依次运行 TopicSendClient 和 TopicRecvClient。

```
[nr@toki examples]$ ant -Dchap=chap6 -Dex=1ps2 run-example
Buildfile: build.xml
...
run-example1ps2:
[java] Begin TopicSendClient, now=1083649621287
[java] Begin sendAsync
```

```
[java] sendAsync, sent text=A text msg, now=1083649621289
[java] End sendAsync
[java] End TopicSendClient
[java] Begin TopicRecvClient, now=1083649625592
[java] Begin recvSync
[java] Timed out waiting for msg
[java] End TopicRecvClient
```

BUILD SUCCESSFUL

Total time: 10 seconds

输出信息表明，topic 订阅者客户（TopicRecvClient）由于超时而没有成功接收发布者发布的消息。

6.1.3 使用持久 topic 的发布/订阅实例

JMS 支持一种消息模型，即处于 P2P 和发布-订阅模型之间的模型。当非活动的发布/订阅客户即使没有监听 topic 时，甚至也想接收到它订阅的、发布到该 topic 的所有消息。使用持久 topic 能够实现这种行为。接下来，依然考虑前面的实例，看看订阅者客户使用持久 topic 是如何保证它能够接收到所有消息的，其中也包括那些客户没有监听该 topic 时所发布的消息。列表 6-5 给出了持久 topic 客户，其中与列表 6-4 的不同重要点用粗体以示差别。

列表 6-5 持久 topic 的 JMS 客户实例

```
package org.jboss.chap6.ex1;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicPublisher;
import javax.jms.TopicSubscriber;
import javax.jms.TopicSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/** A JMS client example program that synchronously receives a
message a Topic

@author Scott.Stark@jboss.org
@version $Revision:$
*/
```

```
public class DurableTopicRecvClient
{
    TopicConnection conn = null;
    TopicSession session = null;
    Topic topic = null;

    public void setupPubSub()
        throws JMSEException, NamingException
    {
        InitialContext iniCtx = new InitialContext();
        Object tmp = iniCtx.lookup("ConnectionFactory");
        TopicConnectionFactory tcf = (TopicConnectionFactory) tmp;
        conn = tcf.createTopicConnection("john", "needle");
        topic = (Topic) iniCtx.lookup("topic/testTopic");
        session = conn.createTopicSession(false,
            TopicSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }

    public void recvSync()
        throws JMSEException, NamingException
    {
        System.out.println("Begin recvSync");
        // Setup the pub/sub connection, session
        setupPubSub();
        // Wait up to 5 seconds for the message
        TopicSubscriber recv = session.createDurableSubscriber(topic,
            "chap6-ex1dtps");
        Message msg = recv.receive(5000);
        if( msg == null )
            System.out.println("Timed out waiting for msg");
        else
            System.out.println("DurableTopicRecvClient.recv, msgt="+msg);
    }

    public void stop() throws JMSEException
    {
        conn.stop();
        session.close();
        conn.close();
    }

    public static void main(String args[]) throws Exception
    {
        System.out.println("Begin DurableTopicRecvClient,
            now="+System.currentTimeMillis());
    }
}
```

```

        DurableTopicRecvClient client = new DurableTopicRecvClient();
        client.recvSync();
        client.stop();
        System.out.println("End DurableTopicRecvClient");
        System.exit(0);
    }

}

```

然后，运行前述的 topic 发布者实例，再运行该持久 topic 订阅者实例。

```

[nr@toki examples]$ ant -Dchap=chap6 -Dex=lpdts run-example
Buildfile: build.xml
...
run-examplelpdts:
[java] Begin DurableTopicSetup
[java] End DurableTopicSetup
[java] Begin TopicSendClient, now=1083649712652
[java] Begin sendAsync
[java] sendAsync, sent text=A text msg, now=1083649712655
[java] End sendAsync
[java] End TopicSendClient
[java] Begin DurableTopicRecvClient, now=1083649716858
[java] Begin recvSync
[java] DurableTopicRecvClient.recv, msgt=org.jboss.mq.SpyTextMessage
[java] Header
[java] jmsDestination : TOPIC.testTopic.DurableSubscriberExample.chap6-ex1dtps
[java] jmsDeliveryMode : 2
[java] jmsExpiration : 0
[java] jmsPriority : 4
[java] jmsMessageID : ID:5-10836497160641
[java] jmsTimeStamp : 1083649716064
[java] jmsCorrelationID: null
[java] jmsReplyTo : null
[java] jmsType : null
[java] jmsRedelivered : false
[java] jmsProperties : {}
[java] jmsPropertiesReadWrite:false
[java] msgReadOnly : true
[java] producerClientId: ID:5
[java] }
[java] Body
[java] text :A text msg, now=1083649712655
[java] }
[java] }
[java] End DurableTopicRecvClient

```


BUILD SUCCESSFUL

Total time: 6 seconds

其中, 该持久 topic 实例中有这样一些问题需要开发者注意。

- 在持久 topic 客户中, TopicConnectionFactory 使用了用户名和密码创建 TopicConnection。同时, 使用 createDurableSubscriber(Topic, String) 方法创建 TopicSubscriber。这些都是持久 topic 订阅者需要满足的。消息服务器需要知道哪个客户在请求持久 topic、持久 topic 订阅名是什么。本文在论述 JBoss 消息配置时将会深入研究持久 topic 设置。
- 在运行 TopicSendClient 之前, 运行了 org.jboss.chap6.ex1.DurableTopicSetup 客户应用。原因在于, 为了使消息服务器能够保存消息, 持久 topic 订阅者必须在过去的某个时间点注册了某订阅。JBoss 支持动态持久 topic 订阅者。其中, DurableTopicSetup 只是简单地创建了持久订阅接收者, 然后离开。但是, 它离开后, 在 “topic/testTopic” 留下了一个活动持久 topic 订阅者, 消息服务器也就知道需要保存发送到该 topic 的任何消息以供后续分发。
- TopicSendClient 在该持久 topic 中并没有变化, 因为这只是持久 topic 订阅者需要注意的事项。
- DurableTopicRecvClient 能够接收到发布在 “topic/testTopic” 上的消息, 其中包括那些客户没有监听该 topic 时所发布的消息。

6.1.4 使用 MDB 的点对点实例

EJB 2.0 规范添加了消息驱动 Bean (Message Driven Bean, MDB)。MDB 是能够异步调用的业务组件。在 EJB 2.0 规范中, 只有 JMS 才能够访问 MDB。列表 6-6 给出了 MDB 实例。其中, 它转换了接收到的 TextMessage 消息, 并将该转换过的消息发送到 queue 中, 该 queue 是通过 JMSReplyTo 头寻找到的。

列表 6-6 处理 TextMessage 的 MDB

```
package org.jboss.chap6.ex2;

import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.ejb.EJBException;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
```

```
import javax.naming.InitialContext;
import javax.naming.NamingException;

/** An MDB that transforms the TextMessages it receives and send the
transformed
messages to the Queue found in the incoming message JMSReplyTo
header.

@author Scott.Stark@jboss.org
@version $Revision:$
*/
public class TextMDB implements MessageDrivenBean, MessageListener
{
    private MessageDrivenContext ctx = null;
    private QueueConnection conn;
    private QueueSession session;

    public TextMDB()
    {
        System.out.println("TextMDB.ctor, this="+hashCode());
    }

    public void setMessageDrivenContext(MessageDrivenContext ctx)
    {
        this.ctx = ctx;
        System.out.println("TextMDB.setMessageDrivenContext,
            this="+hashCode());
    }

    public void ejbCreate()
    {
        System.out.println("TextMDB.ejbCreate, this="+hashCode());
        try
        {
            setupPTP();
        }
        catch(Exception e)
        {
            throw new EJBException("Failed to init TextMDB", e);
        }
    }

    public void ejbRemove()
    {
        System.out.println("TextMDB.ejbRemove, this="+hashCode());
        ctx = null;
    }
}
```

```
        try
        {
            if( session != null )
                session.close();

            if( conn != null )
                conn.close();
        }
        catch(JMSEException e)
        {
            e.printStackTrace();
        }
    }

    public void onMessage(Message msg)
    {
        System.out.println("TextMDB.onMessage, this="+hashCode());
        try
        {
            TextMessage tm = (TextMessage) msg;
            String text = tm.getText() + "processed by: " + hashCode();
            Queue dest = (Queue) msg.getJMSReplyTo();
            sendReply(text, dest);
        }
        catch(Throwable t)
        {
            t.printStackTrace();
        }
    }

    private void setupPTP()
        throws JMSEException, NamingException
    {
        InitialContext iniCtx = new InitialContext();
        Object tmp = iniCtx.lookup("java:comp/env/jms/QCF");
        QueueConnectionFactory qcf = (QueueConnectionFactory) tmp;
        conn = qcf.createQueueConnection();
        session = conn.createQueueSession(false,
            QueueSession.AUTO_ACKNOWLEDGE);
        conn.start();
    }

    private void sendReply(String text, Queue dest)
        throws JMSEException
    {
        System.out.println("TextMDB.sendReply, this="+hashCode()
            +", dest="+dest);
    }
}
```

```

        QueueSender sender = session.createSender(dest);
        TextMessage tm = session.createTextMessage(text);
        sender.send(tm);
        sender.close();
    }
}

```

列表 6-7 给出了该 MDB 的 ejb-jar.xml 和 jboss.xml 部署描述符。

列表 6-7 MDB ejb-jar.xml 和 jboss.xml 描述符

```

// The ejb-jar.xml descriptor
<?xml version="1.0"?>
<!DOCTYPE ejb-jar
PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//
EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd"
>

<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>TextMDB</ejb-name>
      <ejb-class>org.jboss.chap6.ex2.TextMDB</ejb-class>
      <transaction-type>Container</transaction-type>
      <acknowledge-mode>AUTO_ACKNOWLEDGE</acknowledge-mode>
      <message-driven-destination>
        <destination-type>javax.jms.Queue</destination-type>
      </message-driven-destination>
      <res-ref-name>jms/QCF</res-ref-name>
      <resource-ref>
        <res-type>javax.jms.QueueConnectionFactory</res-type>
        <res-auth>Container</res-auth>
      </resource-ref>
    </message-driven>
  </enterprise-beans>
</ejb-jar>

// The jboss.xml descriptor
<?xml version="1.0"?>
<jboss>
  <enterprise-beans>
    <message-driven>
      <ejb-name>TextMDB</ejb-name>
      <destination-jndi-name>queue/B</destination-jndi-name>
      <resource-ref>

```

```
<res-ref-name>jms/QCF</res-ref-name>
<jndi-name>ConnectionFactory</jndi-name>
</resource-ref>
</message-driven>
</enterprise-beans>
</jboss>
```

列表 6-8 给出了另一种 P2P 客户应用。其发送了若干条消息给目的地“queue/B”，并从 queue/A 异步接收 TextMDB 修改过的消息。

列表 6-8 与 TextMDB 交互的 JMS 客户应用

```
package org.jboss.chap6.ex2;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueReceiver;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import EDU.oswego.cs.dl.util.concurrent.CountDown;

/** A complete JMS client example program that sends N
    TextMessages to a Queue B and asynchronously receives the
    messages as modified by TextMDB from Queue A.

    @author Scott.Stark@jboss.org
    @version $Revision:$
    */
public class SendRecvClient
{
    static final int N = 10;
    static CountDown done = new CountDown(N);
    QueueConnection conn;
    QueueSession session;
    Queue queA;
    Queue queB;

    public static class ExListener implements MessageListener
    {
```



```
public void onMessage(Message msg)
{
    done.release();
    TextMessage tm = (TextMessage) msg;
    try
    {
        System.out.println("onMessage, recv text="+tm.getText());
    }
    catch(Throwable t)
    {
        t.printStackTrace();
    }
}

public void setupPTP()
    throws JMSEException, NamingException
{
    InitialContext iniCtx = new InitialContext();
    Object tmp = iniCtx.lookup("ConnectionFactory");
    QueueConnectionFactory qcf = (QueueConnectionFactory) tmp;
    conn = qcf.createQueueConnection();
    queA = (Queue) iniCtx.lookup("queue/A");
    queB = (Queue) iniCtx.lookup("queue/B");
    session = conn.createQueueSession(false,
        QueueSession.AUTO_ACKNOWLEDGE);
    conn.start();
}

public void sendRecvAsync(String textBase)
    throws JMSEException, NamingException, InterruptedException
{
    System.out.println("Begin sendRecvAsync");
    // Setup the PTP connection, session
    setupPTP();
    // Set the async listener for queA
    QueueReceiver recv = session.createReceiver(queA);
    recv.setMessageListener(new ExListener());
    // Send a few text msgs to queB
    QueueSender send = session.createSender(queB);
    for(int m = 0; m < 10; m++)
    {
        TextMessage tm = session.createTextMessage(textBase+"#+m");
        tm.setJMSReplyTo(queA);
        send.send(tm);
        System.out.println("sendRecvAsync, sent text="+tm.getText());
    }
}
```

```

    }

    System.out.println("End sendRecvAsync");
}

public void stop() throws JMSEException
{
    conn.stop();
    session.close();
    conn.close();
}

public static void main(String args[]) throws Exception
{
    System.out.println("Begin
    SendRecvClient,now="+System.currentTimeMillis());
    SendRecvClient client = new SendRecvClient();
    client.sendRecvAsync("A text msg");
    client.done.acquire();
    client.stop();
    System.exit(0);
    System.out.println("End SendRecvClient");
}
}

```

运行该客户端。

```

[nr@toki examples]$ ant -Dchap=chap6 -Dex=2 run-example
Buildfile: build.xml
...
run-example2:
[copy] Copying 1 file to G:\JBoss\jboss-3.2.1\server\default\deploy
[echo] Waiting 5 seconds for deploy...
[java] Begin SendRecvClient, now=1056767530834
[java] Begin sendRecvAsync
[java] sendRecvAsync, sent text=A text msg#0
[java] sendRecvAsync, sent text=A text msg#1
[java] sendRecvAsync, sent text=A text msg#2
[java] sendRecvAsync, sent text=A text msg#3
[java] sendRecvAsync, sent text=A text msg#4
[java] sendRecvAsync, sent text=A text msg#5
[java] sendRecvAsync, sent text=A text msg#6
[java] sendRecvAsync, sent text=A text msg#7
[java] sendRecvAsync, sent text=A text msg#8
[java] sendRecvAsync, sent text=A text msg#9
[java] End sendRecvAsync
[java] onMessage, recv text=A text msg#2processed by: 23715584

```

```
[java] onMessage, recv text=A text msg#6processed by: 322649
[java] onMessage, recv text=A text msg#1processed by: 27534070
[java] onMessage, recv text=A text msg#3processed by: 23966866
[java] onMessage, recv text=A text msg#0processed by: 28532430
[java] onMessage, recv text=A text msg#4processed by: 1734943
[java] onMessage, recv text=A text msg#9processed by: 24814248
[java] onMessage, recv text=A text msg#7processed by: 21845470
[java] onMessage, recv text=A text msg#8processed by: 17243666
[java] onMessage, recv text=A text msg#5processed by: 403211
```

BUILD SUCCESSFUL

Total time: 9 seconds

对应的 JBoss 服务器控制台输出信息如下。

```
19:32:07,209 INFO [MainDeployer] Starting deployment of package: file:/G:/JBoss/
jboss-3.2.1/server/
default/deploy/chap6-ex2.jar
19:32:07,760 INFO [EjbModule] Creating
19:32:07,770 INFO [EjbModule] Deploying TextMDB
19:32:07,810 INFO [MessageDrivenContainer] Creating
19:32:07,810 INFO [MessageDrivenInstancePool] Creating
19:32:07,810 INFO [MessageDrivenInstancePool] Created
19:32:07,810 INFO [JMSContainerInvoker] Creating
19:32:07,820 INFO [JMSContainerInvoker] Created
19:32:07,820 INFO [MessageDrivenContainer] Created
19:32:07,820 INFO [EjbModule] Created
19:32:07,820 INFO [EjbModule] Starting
19:32:07,820 INFO [MessageDrivenContainer] Starting
19:32:07,830 INFO [JMSContainerInvoker] Starting
19:32:07,830 INFO [DLQHandler] Creating
19:32:07,890 INFO [DLQHandler] Created
19:32:08,191 WARN [SecurityManager] No SecurityMetadadata was available for B adding default
security conf
19:32:08,201 INFO [DLQHandler] Starting
19:32:08,201 INFO [DLQHandler] Started
19:32:08,201 INFO [JMSContainerInvoker] Started
19:32:08,201 INFO [MessageDrivenInstancePool] Starting
19:32:08,211 INFO [MessageDrivenInstancePool] Started
19:32:08,211 INFO [MessageDrivenContainer] Started
19:32:08,211 INFO [EjbModule] Started
19:32:08,211 INFO [EJBDeployer] Deployed: file:/G:/JBoss/jboss-3.2.1/server/default/deploy/
chap6-ex2.jar
19:32:08,241 INFO [MainDeployer] Deployed package: file:/G:/JBoss/jboss-3.2.1/server/default/
deploy/chap6-ex2.jar
19:32:12,136 WARN [SecurityManager] No SecurityMetadadata was available for A adding default
security conf
```

```
19:32:12,206 INFO [TextMDB] TextMDB.ctor, this=28532430
19:32:12,216 INFO [TextMDB] TextMDB.ctor, this=27534070
19:32:12,307 INFO [TextMDB] TextMDB.ctor, this=23966866
19:32:12,307 INFO [TextMDB] TextMDB.ctor, this=23715584
19:32:12,307 INFO [TextMDB] TextMDB.setMessageDrivenContext, this=27534070
19:32:12,307 INFO [TextMDB] TextMDB.setMessageDrivenContext, this=28532430
19:32:12,347 INFO [TextMDB] TextMDB.ejbCreate, this=27534070
19:32:12,347 INFO [TextMDB] TextMDB.ejbCreate, this=28532430
19:32:12,347 INFO [TextMDB] TextMDB.setMessageDrivenContext, this=23966866
19:32:12,357 INFO [TextMDB] TextMDB.ejbCreate, this=23966866
19:32:12,377 INFO [TextMDB] TextMDB.setMessageDrivenContext, this=23715584
19:32:12,377 INFO [TextMDB] TextMDB.ejbCreate, this=23715584
19:32:12,387 INFO [TextMDB] TextMDB.ctor, this=1734943
19:32:12,387 INFO [TextMDB] TextMDB.setMessageDrivenContext, this=1734943
19:32:12,387 INFO [TextMDB] TextMDB.ejbCreate, this=1734943
19:32:12,387 INFO [TextMDB] TextMDB.ctor, this=403211
19:32:12,387 INFO [TextMDB] TextMDB.setMessageDrivenContext, this=403211
19:32:12,397 INFO [TextMDB] TextMDB.ejbCreate, this=403211
19:32:12,397 INFO [TextMDB] TextMDB.ctor, this=322649
19:32:12,397 INFO [TextMDB] TextMDB.setMessageDrivenContext, this=322649
19:32:12,397 INFO [TextMDB] TextMDB.ejbCreate, this=322649
19:32:12,437 INFO [TextMDB] TextMDB.ctor, this=21845470
19:32:12,437 INFO [TextMDB] TextMDB.ctor, this=17243666
19:32:12,467 INFO [TextMDB] TextMDB.ctor, this=24814248
19:32:12,467 INFO [TextMDB] TextMDB.setMessageDrivenContext, this=21845470
19:32:12,467 INFO [TextMDB] TextMDB.setMessageDrivenContext, this=24814248
19:32:12,467 INFO [TextMDB] TextMDB.ejbCreate, this=24814248
19:32:12,477 INFO [TextMDB] TextMDB.ejbCreate, this=21845470
19:32:12,487 INFO [TextMDB] TextMDB.setMessageDrivenContext, this=17243666
19:32:12,487 INFO [TextMDB] TextMDB.ejbCreate, this=17243666
19:32:12,537 INFO [TextMDB] TextMDB.onMessage, this=28532430
19:32:12,537 INFO [TextMDB] TextMDB.sendReply, this=28532430, dest=QUEUE.A
19:32:12,537 INFO [TextMDB] TextMDB.onMessage, this=23715584
19:32:12,537 INFO [TextMDB] TextMDB.onMessage, this=23966866
19:32:12,537 INFO [TextMDB] TextMDB.onMessage, this=322649
19:32:12,547 INFO [TextMDB] TextMDB.onMessage, this=27534070
19:32:12,547 INFO [TextMDB] TextMDB.sendReply, this=23715584, dest=QUEUE.A
19:32:12,547 INFO [TextMDB] TextMDB.sendReply, this=23966866, dest=QUEUE.A
19:32:12,547 INFO [TextMDB] TextMDB.sendReply, this=322649, dest=QUEUE.A
19:32:12,567 INFO [TextMDB] TextMDB.sendReply, this=27534070, dest=QUEUE.A
19:32:12,647 INFO [TextMDB] TextMDB.onMessage, this=24814248
19:32:12,647 INFO [TextMDB] TextMDB.sendReply, this=24814248, dest=QUEUE.A
19:32:12,667 INFO [TextMDB] TextMDB.onMessage, this=1734943
19:32:12,677 INFO [TextMDB] TextMDB.sendReply, this=1734943, dest=QUEUE.A
19:32:12,687 INFO [TextMDB] TextMDB.onMessage, this=21845470
19:32:12,687 INFO [TextMDB] TextMDB.sendReply, this=21845470, dest=QUEUE.A
```

```
19:32:12,687 INFO [TextMDB] TextMDB.onMessage, this=17243666
19:32:12,697 INFO [TextMDB] TextMDB.onMessage, this=403211
19:32:12,727 INFO [TextMDB] TextMDB.sendReply, this=17243666, dest=QUEUE.A
19:32:12,727 INFO [TextMDB] TextMDB.sendReply, this=403211, dest=QUEUE.A
```

其中, 开发者需要注意如下事项。

- JMS 客户和其协作的 MDB 之间, 并没有任何关系。客户只是借助于标准 JMS API 发送消息到 queue 中, 并从另一个 queue 接收消息。
- MDB 在 ejb-jar.xml 描述符中指定其监听的 queue 或 topic。其中, queue 或 topic 名必须在 jboss.xml 描述符中指定。在该实例中, MDB 也发送消息给 JMS queue。MDB 可能在 onMessage 回调中充当 queue 发送者或 topic 发布者。
- 客户接收到的消息包含了“processed by: NNN”后缀。其中, NNN 表示处理消息的 MDB 实例的 hashCode 值。它表明, 可能存在多个 MDB 处理提交给目的地的消息。并发处理是 MDB 的优点之一。

6.2 JBoss 消息概述

JBossMQ 是由若干个协作服务组成的, 这些服务为客户应用提供 JMS API 级别的服务。本节将介绍构成 JBossMQ JMS 实现的服务。

6.2.1 调用层

调用层 (Invocation Layer, IL) 服务负责处理客户发送和接收消息的通信协议。JBossMQ 能够同时支持运行不同类型的调用层。所有的调用层都支持双工通信, 即允许客户同时发送和接收消息。IL 只处理消息的具体传输。它们将消息委派给基于 JMX 的 JMS 服务器网关, 即 Invoker。这很类似于分离式 Invoker 是如何借助于不同的传输方式来暴露 EJB 容器的。

各个 IL 服务绑定 JMS 连接工厂到 JNDI 树中的具体位置。客户能够借助于获得 JMS 连接工厂的 JNDI 位置选择待使用的协议。当前, JBossMQ 提供了 6 种不同的调用层, 接下来一一细说。

1. RMI IL (已丢弃)

这个 IL 是基于 Java 远程方法调用 (Remote Method Invocation, RMI) 实现的。由于它是基于标准 RMI 技术, 因此是健壮的 IL, 但是和其他 IL 相比, 就逊色很多。JBoss 以后可能会丢掉它。



这个 IL 会试图建立服务器到客户端的 TCP/IP Socket。因此, 如果客户处于防火墙或有安全性约束以阻止 ServerSocket 的使用时, 则不要使用该 IL。

2. OIL IL（已丢弃）

这个 IL 是基于“优化”IL（“Optimized”IL，OIL）开发的。该 OIL 使用自定义 TCP/IP 协议和性能优良的序列化协议。在没有出现 UIL2 协议之前，JBoss 一直推荐这款基于 Socket 的协议。



这个 IL 会试图建立服务器到客户端的 TCP/IP Socket。因此，如果客户处于防火墙后或有安全性约束以阻止 ServerSocket 的使用时，则不要使用该 IL。

3. UIL IL（已丢弃）

实现的统一调用层（Unified Invocation Layer，UIL）使得处于防火墙或其他约束条件下而不能创建连接的客户能够创建从服务器到客户端的连接。它和 OIL 协议很类似，但是 UIL 使用了多层特性以提供双工通信。这种多层特性会在一个物理 Socket 基础上创建两个虚拟的 Socket。由于引入了多层特性，所以其性能没有 OIL 好。由于出现了 UIL2，因此该调用层现在已经被丢弃。

4. UIL2 IL

统一调用层 Version 2（Unified Version 2 Invocation Layer，UIL2）是 UIL 协议的另一种实现，它也使用客户端和服务器的单个 Socket。然而，不同于其他基于 Socket 的调用层，比如 RMI、UIL 及 OIL，即它们在 Socket 级使用来回阻塞式（blocking round-trip）消息。UIL2 IL 在传输层真正实现了消息异步发送和接收，从而改善了效率和资源利用率。因此，JBoss 推荐这款 Socket 调用层。

5. JVM IL

当 JMS 客户运行在服务器同一 JVM 中时，是不需要跨越 TCP/IP 的，因此 JBoss 引入了 Java 虚拟机调用层。该 IL 运用的直接方法调用，使得服务器能够服务客户请求。从而，增加了操作效率，因为不需要创建 Socket，也不需要相关工作线程的支持。运行在服务器同一 JVM 中的组件，比如 MDB、Servlet、MBean 及 EJB，都应该使用该 IL。

6. HTTP IL

HTTP 调用层（HTTP Invocation Layer，HTTPIL）实现了通过 HTTP 或 HTTPS 协议访问 JBossMQ 服务。该 IL 依赖 deploy/jms/jbossmq-httpil.sar 中部署的 Servlet 来处理 HTTP 通信。当客户处于防火墙后，并且访问端口要求用 HTTP 访问时，则它会有效地访问 JMS 协议。

6.2.2 安全性管理器

JBossMQ SecurityManager 服务给出了访问控制列表（Access Control List，ACL），以保护受访目的地。它和 StateManager 服务协同工作。

6.2.3 目的地管理器

DestinationManager 可以认为是 JBossMQ 提供的核心服务。它掌握了服务器中所有创建的目的地。同时，它还掌握了其他的主要服务，比如 MessageCache、StateManager 及 PersistenceManager。

6.2.4 消息缓存

服务器创建的消息传递给负责内存管理的 MessageCache。当添加给目的地的消息还没有任何接收者时，会增加 JVM 内存容量。除非接收者将这些消息取走，否则它们会一直驻留在主内存中。一旦 MessageCache 注意到 JVM 内容使用已经到达给定上限，则它会将这些消息从内存移动到磁盘的持久化存储器中。MessageCache 使用最近最少使用 (LRU) 算法判断应该移动哪些消息到磁盘中。

6.2.5 状态管理器

StateManager (SM) 负责掌握哪些客户能够登录到服务器及它们的持久订阅名。

6.2.6 持久化管理器

目的地使用 PersistenceManager (PM) 存储那些标记为可以持久化的消息。JBossMQ 提供了若干个持久化管理器的不同实现，但是每个服务器实例只能使用其中一个。开发者可以选择最适合目标机器的持久化管理器。

1. 文件 PM

文件 PM 是 JBossMQ 提供的健壮持久化管理器。它为服务器上的每个目的地创建单独目录，并将持久化消息存储在相应目录的单独文件中。由于需要经常打开和关闭文件，因此其性能很差。

2. 滚动日志 PM

滚动日志 (Rolling Logged) PM 也是基于文件的持久化管理器，但由于它将多个消息存储在单个文件中，减少了打开、关闭多个文件的次数，因此提供了更好的性能。这是一个速度非常快的 PM，但由于它使用了 FileOutputStream.flush()方法调用，因此没有文件 PM 可靠。在某些操作系统和 JVM 中，FileOutputStream.flush()方法并不能保证调用返回时数据已经写入到磁盘中。

3. JDBC2 PM

JDBC2 PM 是 JBossMQ 2.4.x 提供的原始 JDBC PM 的第二版。如今，该版的 PM 做了很多简化和改进工作。该 PM 允许使用 JDBC 将持久化消息存储到关系型数据库中。和其他持久化管理器相比，这个 PM 要求的内存非常少。另外，它还和 MessageCache 紧密

地集成在一起，从而为系统提供有效的持久化。

6.2.7 目的地

目的地是 JBossMQ 服务器中供客户来发送和接收消息的对象。目前，有两种目的地对象类型：queue 和 topic。引用 JBossMQ 创建的目的地被存储在 JNDI 中。

1. queue

在点对点应用模式中客户通常都是使用 queue 的。发送到 queue 的消息只能被一个客户接收，而且只能使用一次。如果多个客户从单个 queue 中接收消息，则将实现消息的负载均衡。在默认情况下，queue 对象存储在 JNDI 子上下文“queue/”中。

2. topic

topic 用于发布/订阅应用模式中。当客户发布消息到 topic 时，则每个已订阅该 topic 的客户都将收到该消息的一份拷贝。分发 topic 消息的方式和电视节目的分发很相似。除非打开了电视机，并在观看电视节目，否则会错过它。类似地，如果客户没有启动、运行，也没有接收到来自 topic 的消息，则它将错过发布到该 topic 的消息。为解决消息丢失的问题，客户可以启动持久订阅。这和 VCR 录制当时不能观看的电视节目很类似，因此再次打开电视机时，能够观看错过的电视节目。

6.3 JBoss 消息配置和 MBean

本节定义 MBean 服务及其属性。其中，这些服务对应于前节介绍的组件。组成 JBossMQ 系统的配置和服务文件包括。

- **conf/jboss-mq-state.xml**: org.jboss.mq.sm.file.DynamicStateManager MBean 需要读取该配置文件。这是存储有效 JMS 用户名和密码的默认安全存储地，以用于认证连接。另外，它还存储活动的持久 topic 订阅。
- **deploy/jms/jbossmq-destinations-service.xml**: 该服务定义了默认的 JMS queue 和 topic 目的地配置，供测试套件单元测试使用。开发者也可以往该文件中添加或删除目的地，或者部署其他带有目的地配置的“*-service.xml”描述符。
- **deploy/jms/jbossmq-service.xml**: 该服务描述符配置核心 JBossMQ MBean，比如 Invoker、SecurityManager、DynamicStateManager 及核心拦截器栈。它也为 MDB 定义了默认的死信队列，即“DLQ”。
- **deploy/jms/jms-ra.rar**: 用于 JMS 供应商的 JCA 资源适配器。
- **deploy/jms/jms-ds.xml**: 它负责配置 JCA 连接工厂和 JMS 供应商 MDB 集成服务。它还将 JBossMQ 设置为 JMS 供应商。
- **deploy/jms/hsqldb-jdbc2-service.xml**: 该服务描述符为 hsqldb 数据库配置 DestinationManager、MessageCache 及 JDBC2 PersistenceManager。
- **deploy/jms/jvm-il-service.xml**: 该服务描述符配置提供 JVM IL 传输的 JVM

ServerILService。

- **deploy/jms/oil-service.xml**: 该服务描述符配置提供 OIL 传输的 OILServerILService。该 IL 的 queue 和 topic 连接工厂绑定在 JNDI 名“ConnectionFactory”中。
- **deploy/jms/oil2-service.xml**: OIL 传输的实验性版本。建议开发者不要使用, JBoss 后续发布版将会删除它。
- **deploy/jms/rmi-il-service.xml**: 该服务描述符配置提供 RMI IL 的 RMIServerILService。该 IL 的 queue 和 topic 连接工厂绑定在 JNDI 名“RMIServerILService”中。
- **deploy/jms/uil2-service.xml**: 该服务描述符配置提供 UIL2 传输的 UILServerILService。该 IL 的 queue 和 topic 连接工厂绑定在 JNDI 名“UIL2ConnectionFactory”和“UILConnectionFactory”中。其中, JNDI 名, 即“UILConnectionFactory”的目的在于替代已丢弃的第一版 UIL 服务。

接下来, 本文将讨论相关的 MBean 服务。

6.3.1 org.jboss.mq.il.jvm.JVMServerILService

org.jboss.mq.il.jvm.JVMServerILService MBean 用于配置 JVM IL。可配置的属性如下。

- **Invoker**: 该属性指定 JMS 入口服务的 JMX ObjectName。其中, 该服务用于传递请求给 JMS 服务器。开发者借助于<depends optional=attribute-name="Invoker">标签设置该属性。除非变更了该入口服务, 否则开发者不要随便改动“jboss.mq:service=Invoker”设置。
- **ConnectionFactoryJNDIRef**: 该 IL 的 ConnectionFactory 绑定的 JNDI 位置。
- **XAConnectionFactoryJNDIRef**: 该 IL 的 XAConnectionFactory 绑定的 JNDI 位置。
- **PingPeriod**: 客户发送 ping 消息到服务器的周期 (单位: 毫秒), 以判断连接是否还有效。如果设为 0, 则不发送 ping 消息。既然 JVM IL 连接不可能失效, 因此建议开发者将该值一直设为“0”。

6.3.2 org.jboss.mq.il.rmi.RMIServerILService (已丢弃)

org.jboss.mq.il.rmi.RMIServerILService 用于配置 RMI IL。可配置的属性如下。

- **Invoker**: 该属性指定 JMS 入口服务的 JMX ObjectName。其中, 该服务用于传递请求给 JMS 服务器。开发者借助于<depends optional=attribute-name="Invoker">标签设置该属性。除非变更了该入口服务, 否则开发者不要随便改动“jboss.mq:service=Invoker”设置。
- **ConnectionFactoryJNDIRef**: 该 IL 的 ConnectionFactory 绑定的 JNDI 位置。
- **XAConnectionFactoryJNDIRef**: 该 IL 的 XAConnectionFactory 绑定的 JNDI 位置。
- **PingPeriod**: 客户发送 ping 消息到服务器的周期 (单位: 毫秒), 以判断连接是否还有效。如果设为 0, 则不发送 ping 消息。

6.3.3 org.jboss.mq.il.oil.OILServerILService（已丢弃）

org.jboss.mq.il.oil.OILServerILService 用于配置 OIL IL。可配置的属性如下。

- **Invoker:** 该属性指定 JMS 入口服务的 JMX ObjectName。其中，该服务用于传递请求给 JMS 服务器。开发者借助于<depends optional=attribute-name="Invoker">标签设置该属性。除非变更了该入口服务，否则开发者不要随便改动“jboss.mq:service=Invoker”设置。
- **ConnectionFactoryJNDIRef:** 该 IL 的 ConnectionFactory 绑定的 JNDI 位置。
- **XAConnectionFactoryJNDIRef:** 该 IL 的 XAConnectionFactory 绑定的 JNDI 位置。
- **PingPeriod:** 客户发送 ping 消息到服务器的周期（单位：毫秒），以判断连接是否还有效。如果设为 0，则不发送 ping 消息。
- **ReadTimeout:** 传递给 UIL2 Socket 的 SoTimeout 值（单位：毫秒）。它能够检测那些没有响应的和不能够接收 ping 消息的过期 Socket。请开发者注意，该值应该大于 PingPeriod 属性值。
- **ServerBindPort:** 该 IL 协议监听的端口。默认值为 0，即随机选择一个可用端口。
- **BindAddress:** 该 IL 监听的具体地址。它能够用于存在多个主机地址的机器上，即为 java.net.ServerSocket 提供监听地址。但只有其中一个地址接受客户的请求。
- **EnableTcpNoDelay:** 如果设为 true，则使 TcpNoDelay 生效。既然只要一请求刷新（flush）就会发送 TCP/IP 包，所以改善了请求响应的时间。否则，操作系统可能会缓存请求包，然后创建更大的 IP 包。
- **ServerSocketFactory:** javax.net.ServerSocketFactory 实现的类名，用于创建服务 java.net.ServerSocket。如果没有指定，则开发者通过方法 javax.net.ServerSocketFactory.getDefault()获得。
- **ClientSocketFactory:** javax.net.SocketFactory 实现的类名，供客户使用。如果没有指定，则开发者通过 javax.net.SocketFactory.getDefault()获得其默认值。
- **SecurityDomain:** 指定和 JBoss SSL Socket 工厂一起使用的安全性域名。它是安全性管理器实现的 JNDI 名。在 8.3 节中的“再次生效 JBoss 安全性声明”小节中的 jboss.xml 和 jboss-web.xml 描述符阐述了 security-domain 元素。

6.3.4 org.jboss.mq.il.uil.UILServerILService（已丢弃）

org.jboss.mq.il.uil.UILServerILService 用于配置 UIL IL。需要开发者注意的是，JBoss 3.2.2 默认发布版已经删除了该服务，但通过 docs/examples/jca 目录还是能够找到相应的实例配置文件。

UILServerILService 的可配置属性如下。

- **Invoker:** 该属性指定 JMS 入口服务的 JMX ObjectName。其中，该服务用于传递请求给 JMS 服务器。开发者借助于<depends optional=attribute-name="Invoker">

标签设置该属性。除非变更了该入口服务，否则开发者不要随便改动“jboss.mq:service=Invoker”设置。

- **ConnectionFactoryJNDIRef**: 该 IL 的 ConnectionFactory 绑定的 JNDI 位置。
- **XAConnectionFactoryJNDIRef**: 该 IL 的 XAConnectionFactory 绑定的 JNDI 位置。
- **PingPeriod**: 客户发送 ping 消息到服务器的周期（单位：毫秒），以判断连接是否还有效。如果设为 0，则不发送 ping 消息。
- **ServerBindPort**: 该 IL 协议监听的端口。默认值为 0，即随机选择一个可用端口。
- **BindAddress**: 该 IL 监听的具体地址。它能够用于存在多个主机地址的机器上，即为 java.net.ServerSocket 提供监听地址。但只有其中一个地址接受客户的请求。
- **EnableTcpNoDelay**: 如果设为 true，则使 TcpNoDelay 生效。既然只要一请求刷新（flush）就会发送 TCP/IP 包，所以改善了请求响应的时间。否则，操作系统可能会缓存请求包，然后创建更大的 IP 包。
- **ServerSocketFactory**: javax.net.ServerSocketFactory 实现的类名，用于创建服务 java.net.ServerSocket。如果没有指定，则开发者通过方法 javax.net.ServerSocketFactory.getDefault() 获得。
- **ClientSocketFactory**: javax.net.SocketFactory 实现的类名，供客户使用。如果没有指定，则开发者通过 javax.net.SocketFactory.getDefault() 获得其默认值。
- **SecurityDomain**: 指定和 JBoss SSL Socket 工厂一起使用的安全性域名。它是安全性管理器实现的 JNDI 名。在 8.3 中的“再次生效 JBoss 安全性声明”小节中的 jboss.xml 和 jboss-web.xml 描述符阐述了 security-domain 元素。

6.3.5 org.jboss.mq.il.util2.UILServerILService

org.jboss.mq.il.util2.UILServerILService 用于配置 UIL2 IL。可配置的属性如下。

- **Invoker**: 该属性指定 JMS 入口服务的 JMX ObjectName。其中，该服务用于传递请求给 JMS 服务器。开发者借助于<depends optional=attribute-name="Invoker">标签设置该属性。除非变更了该入口服务，否则开发者不要随便改动“jboss.mq:service=Invoker”设置。
- **ConnectionFactoryJNDIRef**: 该 IL 的 ConnectionFactory 绑定的 JNDI 位置。
- **XAConnectionFactoryJNDIRef**: 该 IL 的 XAConnectionFactory 绑定的 JNDI 位置。
- **PingPeriod**: 客户发送 ping 消息到服务器的周期（单位：毫秒），以判断连接是否还有效。如果设为 0，则不发送 ping 消息。
- **ReadTimeout**: 传递给 UIL2 Socket 的 SoTimeout 值（单位：毫秒）。它能够检测那些没有响应的和不能够接收 ping 消息的过期 Socket。需要开发者注意的是，该值应该大于 PingPeriod 属性值。
- **BufferSize**: 字节大小，用于基本 Socket 流中的缓存大小。其值对应于 java.io.BufferedOutputStream 的缓存大小。
- **ChunkSize**: 流监听器通知之间的字节大小。统一调用层 Version 2 使用支持 heartbeat

的 `org.jboss.util.stream.NotifyingBufferedOutputStream` 和 `NotifyingBufferedInputStream` 实现。其中, 该 `heartbeat` 是由读出或写入流的数据触发的。无论如何, `ChunkSize` 字节都是读出或写入到流中。当大数据的读写操作需要的时间比 `PingPeriod` 长时, 它能够充当 `ping` 或保持通知的有效性。

- **ServerBindPort:** 该 IL 协议监听的端口。默认值为 0, 即随机选择一可用端口。
- **BindAddress:** 该 IL 监听的具体地址。它能够用于存在多个主机地址的机器上, 即为 `java.net.ServerSocket` 提供监听地址。但只有其中一个地址接受客户请求。
- **EnableTcpNoDelay:** 如果设为 `true`, 则使 `TcpNoDelay` 生效。既然只要一请求刷新 (`flush`) 就会发送 TCP/IP 包, 所以改善了请求响应的时间。否则, 操作系统可能会缓存请求包, 然后创建更大的 IP 包。
- **ServerSocketFactory:** `javax.net.ServerSocketFactory` 实现的类名, 用于创建服务 `java.net.ServerSocket`。如果没有指定, 则开发者通过方法 `javax.net.ServerSocketFactory.getDefault()` 获得。
- **ClientSocketFactory:** `javax.net.SocketFactory` 实现的类名, 供客户使用。如果没有指定, 则开发者通过 `javax.net.SocketFactory.getDefault()` 获得其默认值。
- **SecurityDomain:** 指定和 JBoss SSL Socket 工厂一起使用的安全性域名。它是安全性管理器实现的 JNDI 名。在 8.3 中的“再次生效 JBoss 安全性声明”小节中的 `jboss.xml` 和 `jboss-web.xml` 描述符阐述了 `security-domain` 元素。

为 SSL 配置 IL

UIL2 和 OIL 服务通过自定义 Socket 工厂能够支持使用 SSL。其中, 这些工厂使用 IL 服务关联的安全性域集成了 JSSE。列表 6-9 给出了 UIL2 服务描述符片段的实例, 它描述了自定义 JBoss SSL Socket 工厂的使用。

列表 6-9 使用 SSL 的 UIL 配置实例片段

```
<mbean code="org.jboss.mq.il.util2.UILServerILService"
  name="jboss.mq:service=InvocationLayer,type=HTTPSUIL2">
  <depends optional-attribute-name="Invoker">jboss.mq:service=Invoker
</depends>
  <attribute name="ConnectionFactoryJNDIRef">SSLConnectionFactory
</attribute>
  <attribute name="XAConnectionFactoryJNDIRef">SSLXAConnectionFactory
</attribute>
...
  <!-- SSL Socket Factories -->
  <attribute name="ClientSocketFactory">
    org.jboss.security.ssl.ClientSocketFactory
  </attribute>
  <attribute name="ServerSocketFactory">
    org.jboss.security.ssl.DomainServerSocketFactory
  </attribute>
  <!-- Security domain - see below -->
  <attribute name="SecurityDomain">java:/jaas/SSL</attribute>
```

```

</mbean>

<!-- Configures the keystore on the "SSL" security domain
This mbean is better placed in conf/jboss-service.xml where it
can be used by other services, but it will work from anywhere.
Use keytool from the sdk to create the keystore.
-->
<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
  name="jboss.security:service=JaasSecurityDomain,domain=SSL">
  <!-- This must correlate with the java:/jaas/SSL above -->
  <constructor>
    <arg type="java.lang.String" value="SSL"/>
  </constructor>
  <!-- The location of the keystore resource: loads from the
  classpath and the server conf dir is a good default -->
  <attribute name="KeyStoreURL">resource:uil2.keystore</attribute>
  <attribute name="KeyStorePass">changeme</attribute>
</mbean>

```

6.3.6 org.jboss.mq.il.http.HTTPServerILService

org.jboss.mq.il.http.HTTPServerILService 用于管理 HTTP(S) IL。该 IL 允许以 HTTP 或 HTTPS 连接方式使用 JMS 服务。其依赖于 deploy/jms/jbossmq-httpil.sar 部署的 Servlet 处理 HTTP 通信。可配置的属性如下。

- **Timeout:** 客户 HTTP 请求等待消息的默认超时时间（单位：秒）。在客户端，开发者通过设置系统属性 org.jboss.mq.il.http.timeout 能够覆盖默认超时时间。
- **RestInterval:** 每次请求过后，客户睡眠的时间（单位：秒）。默认值为 0，但也可以同 Timeout 一起使用而实现完全基于计时的轮流机制。比如，将 Timeout 设置为 0，RestInterval 为 60，则能完成短时请求操作。随后，客户将发送非阻塞请求到服务器，返回任何可用消息，然后在发送下一个请求前先睡眠 60 秒。同 Timeout 一样，通过显式地指定 org.jboss.mq.il.http.restinterval 能够覆盖 RestInterval 值。
- **URL:** 设置 Servlet 的 URL。该值由若干个值集合构成（比如，URLPrefix、URLSuffix、URLPort 等）。它可能真的是 URL，或者属性名，客户端将通过调用 System.getProperty(propertyname) 解析该属性名。如果没有指定 URL，将通过 URLPrefix+URLHostName+": "+URLPort+"/"+URLSuffix 组成。
- **URLPrefix:** Servlet URL 前缀部分。
- **URLHostName:** Servlet URL 主机名部分。
- **URLPort:** URL 的端口部分。
- **URLSuffix:** URL 的后缀部分。
- **UseHomeName:** 标志位。如果为 true，则 URLHostName 属性取值将通过 InetAddress.getLocalHost().getHostName() 获得。如果为 false，则 URLHostName

属性取值将通过 `InetAddress.getLocalHost().getHostAddress()` 获得。

6.3.7 org.jboss.mq.server.jmx.Invoker

`org.jboss.mq.server.jmx.Invoker` 借助于拦截器栈传递 IL 请求到目的地管理器服务。可配置的属性如下：

- **NextInterceptor**: 下一个拦截器的 JMX ObjectName。所有的拦截器都是使用该属性创建拦截器栈。链中的末尾拦截器应该是 `DestinationManager`。开发者借助于 `<depends optional-attribute-name="NextInterceptor">` 标记可以设置该属性。

6.3.8 org.jboss.mq.server.jmx.InterceptorLoader

`org.jboss.mq.server.jmx.InterceptorLoader` 用于装载常见拦截器，并将其作为拦截器栈的一部分。该 MBean 通常都是用于装载自定义拦截器，比如自定义拦截器 `org.jboss.mq.server.TracingInterceptor`，它借助于日志消息能够有效地记录所有客户请求。可配置的属性如下。

- **NextInterceptor**: 下一个拦截器的 JMX ObjectName。所有的拦截器都是使用该属性创建拦截器栈。链中的末尾拦截器应该是 `DestinationManager`。开发者借助于 `<depends optional-attribute-name="NextInterceptor">` 标记可以设置该属性。
- **InterceptorClass**: 待装载的拦截器类名，它是拦截器栈的组成部分。指定的类名必须继承于 `org.jboss.mq.server.JMSServerInterceptor` 类。

6.3.9 org.jboss.mq.sm.file.DynamicStateManager

`org.jboss.mq.sm.file.DynamicStateManager` MBean 是默认状态管理器，并且它还被分配给 `DestinationManager` 服务。它管理 XML 形式的用户安全性存储源，以提供认证、授权及持久订阅者信息。可配置的属性如下。

- **StateFile**: 该文件用于存储状态信息，比如创建的持久订阅。它是服务器完成数据读写操作的 XML 文件，图 6-1 给出了其内容模型。当服务器运行时，不要编辑该 XML 文件。默认值为 `conf/jbossmq-state.xml` 文件。
 - **User/Name**: 对应于 `Connection.createConnection(username,password)` 方法的用户名。
 - **User/Password**: 对应于 `Connection.createConnection(username,password)` 方法的密码。
 - **User/Id**: 通过用户名创建的连接所关联的 ClientID，即限制了客户只能有单个活动连接。
 - **DurableSubscriptions/DurableSubscription/ClientID**: 关联持久订阅的惟一客户连接 id。
 - **DurableSubscriptions/DurableSubscription/Name**: 持久订阅名。它是方法 `TopicSession.createDurableSubscriber(Topic,name)` 的 name 参数值。
 - **DurableSubscriptions/DurableSubscription/TopicName**: 当前持久订阅关联的 topic 名。

- **HasSecurityManager:** boolean 标志位, 表明 JAAS SecurityManager 服务是否配置成核心 JMS 服务器拦截器栈的组成部分。如果为 false, 则该服务完成连接认证。默认值为 true。

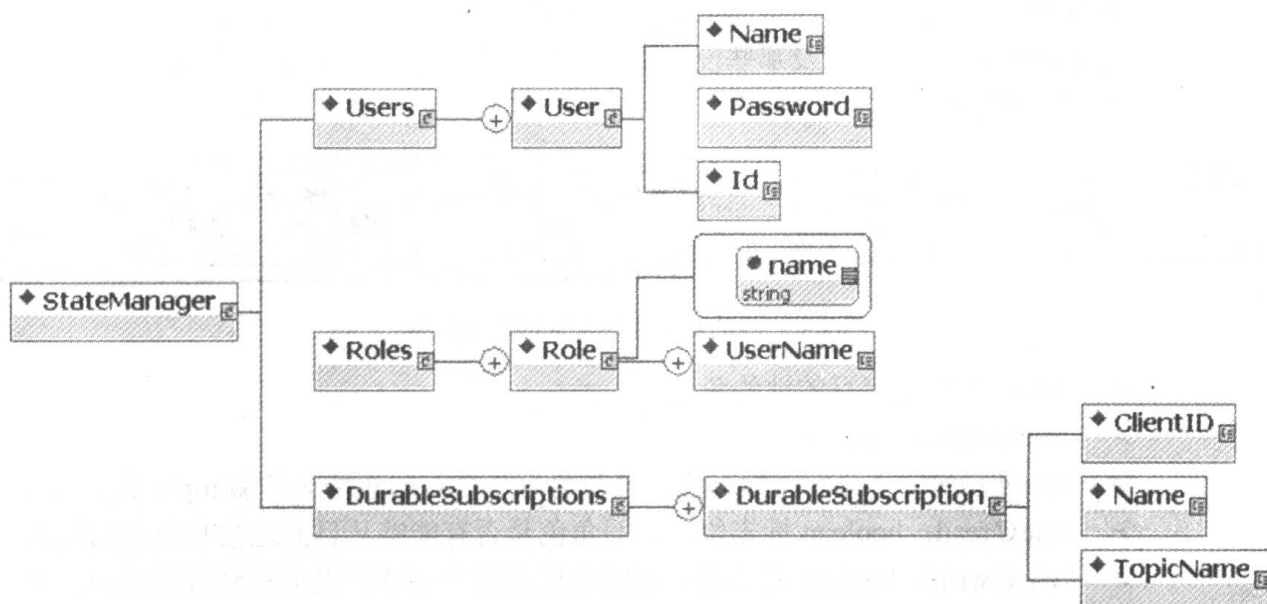


图 6-1 jboss-state.xml 内容模型

6.3.10 org.jboss.mq.security.SecurityManager

如果 org.jboss.mq.security.SecurityManager 是拦截器栈的组成部分, 则它将加强分配给目的地的访问控制列表。SecurityManager 使用了 JAAS, 因此要求在 JBoss login-config.xml 中设置应用策略。列表 6-10 给出了其默认配置。

列表 6-10 用于 JBossMQ 的默认 login-config.xml 配置

```
<application-policy name = "jbossmq">
  <authentication>
    <login-module code = "org.jboss.mq.sm.file.DynamicLoginModule"
      flag = "required">
      <module-option name="unauthenticatedIdentity">guest
    </module-option>
    <module-option name="sm.objectname">
      jboss.mq:service=StateManager
    </module-option>
    </login-module>
  </authentication>
</application-policy>
```

它通过 org.jboss.mq.sm.file.DynamicLoginModule 将 DynamicStateManager jbossmq-state.xml 安全性存储源集成到基于 JAAS 的框架中。该配置还将任何未认证的 JBossMQ 客户映射为“guest”角色。

SecurityManager 的可配置的属性如下。

- **NextInterceptor**: 下一个拦截器的 JMX ObjectName。所有的拦截器都是使用该属性创建拦截器栈。链中的末尾拦截器应该是 DestinationManager。开发者借助于<depends optional-attribute-name="NextInterceptor">标记可以设置该属性。
- **DefaultSecurityConfig**: 该元素为目的地指定默认安全性配置信息。它适合于临时 queue 和 topic，以及那些没有指定具体安全性配置的 queue 和 topic。图 6-2 给出了该元素的内容模型。

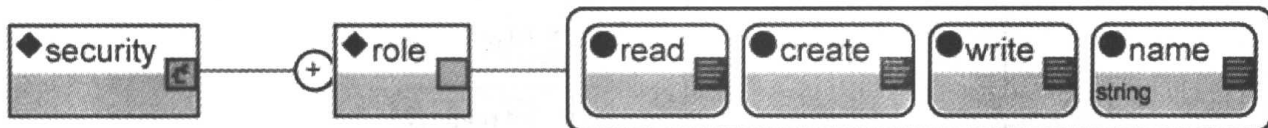


图 6-2 目的地安全性配置的内容模型

- **role**: 有权访问目的地的角色。
- **role@name**: 定义角色名。
- **role@create**: boolean 标志位，表明角色是否有创建持久订阅 topic 的能力。
- **role@read**: boolean 标志位，表明角色是否有能够从目的地接收消息的能力。
- **role@write**: boolean 标志位，表明角色是否有能够发送消息到目的地的能力。
- **SecurityDomain**: 用于认证和基于角色的授权。它是安全性管理器实现的 JNDI 名。在 8.3 节中的“再次生效 JBoss 安全性声明”小节中的 jboss.xml 和 jboss-web.xml 描述符阐述了 security-domain 元素。然而有一点请开发者注意，该属性值不能有“java:jaas”前缀，因为该前缀当前是一假定、硬编码值。

通过 XML 文件维护认证和授权信息，可能不是开发者所乐意的。当然，开发者可以使用任何标准安全性存储源，比如数据库或 LDAP 服务器。只要简单地更新 JAAS login-config.xml，并完成相同用户名到密码、用户到角色的映射及 DynamicStateManager 的更新，如表 6-1 和表 6-2 所示。比如，为使用 JDBC 数据库，开发者使用如下实例数据库和 login-config.xml 入口应该可以达到目的，如列表 6-11 所示。

表 6-1 “JMSPasswords”用户名到密码实例表

用户名	密码
Jduke	theduke

表 6-2 “JMSRoles”用户名到角色实例表

用户名	角色
jduke	create
jduke	read
jduke	write

列表 6-11 用于 JBossMQ 的另一 login-config.xml 配置

```
<application-policy name = "jbossmq">
  <authentication>
    <login-module code =
```

```

    "org.jboss.security.auth.spi.DatabaseServerLoginModule"
    flag = "required">
<module-option name="unauthenticatedIdentity">guest
</module-option>
<module-option name="dsJndiName">java:/DefaultDS
</module-option>
<module-option name="principalsQuery">
    select password from JMSPasswords where username = ?
</module-option>
<module-option name="rolesQuery">
    select role, "Roles" from JMSRoles where username= ?
</module-option>
</login-module>
</authentication>
</application-policy>

```

在 8.4.6 小节中的“4. org.jboss.security.auth.spi.DatabaseServerLoginModule”有 DatabaseServerLoginModule 的完整描述。

6.3.11 org.jboss.mq.server.jmx.DestinationManager

org.jboss.mq.server.jmx.DestinationManager 必须为拦截器栈中的末尾拦截器。可配置的属性如下。

- **PersistenceManager**: 服务器使用的持久化管理器服务的 JMX ObjectName。开发者借助于<depends optional-attribute-name="PersistenceManager">XML 标签能够设置该属性。
- **StateManager**: 服务器使用的状态管理器服务的 JMX ObjectName。开发者借助于<depends optional-attribute-name="StateManager">XML 标签能够设置该属性。
- **MessageCache**: 服务器使用的消息缓存服务的 JMX ObjectName。开发者借助于<depends optional-attribute-name="MessageCache">XML 标签能够设置该属性。

其他支持监控目的的只读属性和操作如下。

- **ClientCount**: 连接到服务器的客户数量。
- **Clients**: Clients 属性指定连接到服务器的客户实例，通过如下形式表达：
java.util.Map<org.jboss.mq.ConnectionToken,org.jboss.mq.server.ClientConsumer>。
- **MessageCounter**: org.jboss.mq.server.MessageCounter 实例数组，为 JMS 目的地提供统计资料。
- **String listMessageCount()**: 该操作生成 HTML 表，包含如下内容。
 - **Type**: 表明目的地类型，即 queue 或 topic。
 - **Name**: 目的地名。
 - **Subscription**: 某 topic 的订阅 id。
 - **Durable**: boolean 标志位，表明某 topic 订阅是否是持久的。
 - **Count**: 分发到目的地的消息数量。
 - **CountDelta**: 自从上次访问 Count 以来，消息数量的变化。

- Depth: 目的地中的消息数量。
- DepthDelta: 自从上次访问 Depth 以来, 目的地中消息数量的变化。
- Last Add: 格式为 DataFormat.SHORT/DataFormat.MEDIUM 的日期/时间字符串, 表明最近一次添加消息到目的地的时间。
- void resetMessageCounter(): 清零所有的目的地计数器和最近添加的时间。

6.3.12 org.jboss.mq.server.MessageCache

服务器使用 org.jboss.mq.server.MessageCache MBean 决定何时将消息移动到辅助存储源中。可配置的属性如下。

- **CacheStore**: 充当缓存存储源服务的 JMX ObjectName。MessageCache 使用缓存存储源将消息移动到持久存储源中。其具体取值要看使用的具体持久化管理器类型。开发者借助于<depends optional-attribute-name="CacheStore">XML 标签能够设置它。
- **HighMemoryMark**: 在 MessageCache 开始移动消息到辅助存储源前, 必须达到的 JVM 堆的内存数量 (单位: MB)。
- **MaxMemoryMark**: JVM 堆内存的最大数量 (单位: MB), MessageCache 称其为最大内存标记。当内存使用达到最大内存标记时, MessageCache 将移动消息到持久化存储源中, 使得保存在内存中的消息数量接近 0。

其他提供统计信息的只读属性如下。

- CacheHits: 内存中消息被请求的次数。
- CacheMisses: 消息被请求的次数。其中, 由于这些消息不在内存中, 所以需要从持久化存储源中获得这些消息。
- HardRefCacheSize: 必须待在内存中的消息数量, 称为硬引用 (hard reference)。
- SoftRefCacheSize: 已经完成持久化, 但还逗留在内存中的消息数量, 称为软引用 (soft reference)。主要是考虑到垃圾收集器还未来得及释放空间。
- TotalCacheSize: 缓存管理的消息总数。

6.3.13 org.jboss.mq.pm.file.CacheStore

当目标 JBoss 服务器使用文件或滚动日志持久化管理器时, 开发者应该使用 org.jboss.mq.pm.file.CacheStore MBean 作为 MessageCache 服务的缓存存储源。可配置的属性如下。

- **DataDirectory**: 为 MessageCache 存储消息的目录。

6.3.14 org.jboss.mq.pm.file.PersistenceManager

如果希望使用文件 PM, 则开发者应该将 org.jboss.mq.pm.file.PersistenceManager 作为持久化管理器, 以分配给 DestinationManager。可配置的属性如下。

- **MessageCache**: 分配给 DestinationManager 的 MessageCahe 的 JMX ObjectName。开发者借助于<depends optional-attribute-name="MessageCache">XML 标签能够设置该属性。

- **DataDirectory**: 用于存储持久化消息的目录。

6.3.15 org.jboss.mq.pm.rollinglogged.PersistenceManager

如果希望使用滚动日志持久化管理器，则开发者应该将 MBean 服务，即 org.jboss.mq.pm.rollinglogged.PersistenceManager，作为持久化管理器，以分配给 DestinationManager。可配置的属性如下。

- **DataDirectory**: 用于存储持久化消息的目录。
- **RollOverSize**: 到达日志轮回的最大消息数量。

6.3.16 org.jboss.mq.pm.jdbc2.PersistenceManager

如果希望将消息存储在数据库中，则开发者应该将如下 MBean 服务，即 org.jboss.mq.pm.jdbc2.PersistenceManager，作为持久化管理器，以分配给 DestinationManager。该 PM 已经测试通过 HypersonicSQL、MS SQL、Oracle、MySQL 及 Postgres 数据库。可配置的属性如下。

- **MessageCache**: MessageCache 属性表明，已分配给 DestinationManager 的 MessageCache 的 JMX ObjectName。开发者借助于<depends optional-attribute-name="MessageCache">标签能够设置该属性。
- **ConnectionManager**: 用户获取 JDBC 连接的 JCA 数据源的 JMX ObjectName。开发者借助于<depends optional-attribute-name="DataSource">标签能够设置该属性。用户还可能需要另外一个<depends>XML 标签，以等待该 PM 启动前能够启动数据源连接管理器服务。
- **ConnectionRetryAttempts**: 整数，PM 使用它重新获得 JDBC 存储源的连接。每次连接失败和下次连接操作之间有 1 500 毫秒的延迟。该数应该大于等于 1，其默认值为 5。
- **SqlProperties**: 属性列表，用于定义 SQL 查询和其他 JDBC2 持久化管理器选项。如果不希望使用 HypersonicSQL，而想使用其他数据库，则开发者需要调整这些属性值。列表 6-12 给出了用于 Hypersonic 数据库的默认配置。列表 6-13 给出了用于 Oracle 数据库的另一配置。其他的实例，开发者可以在 JBoss 发布版的 docs/examples/jms 目录下找到。

列表 6-12 默认 JDBC2 持久化管理器 SqlProperties

```
<attribute name="SqlProperties">
  BLOB_TYPE=OBJECT_BLOB
  INSERT_TX = INSERT INTO JMS_TRANSACTIONS (TXID) values(?)
  INSERT_MESSAGE = INSERT INTO JMS_MESSAGES (MESSAGEID, DESTINATION,
    MESSAGEBLOB, TXID, TXOP) VALUES(?,?,?, ?,?)
  SELECT_ALL_UNCOMMITTED_TXS = SELECT TXID FROM JMS_TRANSACTIONS
  SELECT_MAX_TX = SELECT MAX(TXID) FROM JMS_MESSAGES
  SELECT_MESSAGES_IN_DEST = SELECT MESSAGEID, MESSAGEBLOB FROM JMS_MESSAGES WHERE DESTINATION=?
  SELECT_MESSAGE = SELECT MESSAGEID, MESSAGEBLOB FROM JMS_MESSAGES WHERE MESSAGEID=? AND DESTINATION=?
  MARK_MESSAGE = UPDATE JMS_MESSAGES SET TXID=?, TXOP=? WHERE MESSAGEID=? AND DESTINATION=?
</attribute>
```

```

UPDATE_MESSAGE = UPDATE JMS_MESSAGES SET MESSAGEBLOB=? WHERE MESSAGEID=? AND DESTINATION=?
UPDATE_MARKED_MESSAGES = UPDATE JMS_MESSAGES SET TXID=?, TXOP=? WHERE TXOP=?
UPDATE_MARKED_MESSAGES_WITH_TX = UPDATE JMS_MESSAGES SET TXID=?, TXOP=? WHERE TXOP=? AND TXID=?
DELETE_MARKED_MESSAGES_WITH_TX = DELETE FROM JMS_MESSAGES WHERE TXID IN (SELECT TXID FROM
    JMS_TRANSACTIONS) AND TXOP=?
DELETE_TX = DELETE FROM JMS_TRANSACTIONS WHERE TXID = ?
DELETE_MARKED_MESSAGES = DELETE FROM JMS_MESSAGES WHERE TXID=? AND TXOP=?
DELETE_MESSAGE = DELETE FROM JMS_MESSAGES WHERE MESSAGEID=? AND
    DESTINATION=?
CREATE_MESSAGE_TABLE = CREATE TABLE JMS_MESSAGES ( MESSAGEID INTEGER NOT NULL, DESTINATION VARCHAR(255)
    NOT NULL, TXID INTEGER, TXOP CHAR(1), MESSAGEBLOB OBJECT, PRIMARY KEY (MESSAGEID, DESTINATION) )
CREATE_TX_TABLE = CREATE TABLE JMS_TRANSACTIONS ( TXID INTEGER )
</attribute>

```

列表 6-13 用于 Oracle 的 JDBC2 持久化管理器 SqlProperties 实例

```

<attribute name="SqlProperties">
    BLOB_TYPE=BINARYSTREAM_BLOB
    INSERT_TX = INSERT INTO JMS_TRANSACTIONS (TXID) values(?)
    INSERT_MESSAGE = INSERT INTO JMS_MESSAGES (MESSAGEID, DESTINATION,
        MESSAGEBLOB, TXID, TXOP) VALUES(?,?,?,?)
    SELECT_ALL_UNCOMMITTED_TXS = SELECT TXID FROM JMS_TRANSACTIONS
    SELECT_MAX_TX = SELECT MAX(TXID) FROM JMS_MESSAGES
    SELECT_MESSAGES_IN_DEST = SELECT MESSAGEID, MESSAGEBLOB FROM JMS_MESSAGES WHERE DESTINATION=?
    SELECT_MESSAGE = SELECT MESSAGEID, MESSAGEBLOB FROM JMS_MESSAGES WHERE MESSAGEID=? AND DESTINATION=?
    MARK_MESSAGE = UPDATE JMS_MESSAGES SET TXID=?, TXOP=? WHERE MESSAGEID=? AND DESTINATION=?
    UPDATE_MESSAGE = UPDATE JMS_MESSAGES SET MESSAGEBLOB=? WHERE MESSAGEID=? AND DESTINATION=?
    UPDATE_MARKED_MESSAGES = UPDATE JMS_MESSAGES SET TXID=?, TXOP=? WHERE TXOP=?
    UPDATE_MARKED_MESSAGES_WITH_TX = UPDATE JMS_MESSAGES SET TXID=?, TXOP=? WHERE TXOP=? AND TXID=?
    DELETE_MARKED_MESSAGES_WITH_TX = DELETE FROM JMS_MESSAGES WHERE TXID IN (SELECT TXID FROM
        JMS_TRANSACTIONS) AND TXOP=?
    DELETE_TX = DELETE FROM JMS_TRANSACTIONS WHERE TXID = ?
    DELETE_MARKED_MESSAGES = DELETE FROM JMS_MESSAGES WHERE TXID=? AND TXOP=?
    DELETE_MESSAGE = DELETE FROM JMS_MESSAGES WHERE MESSAGEID=? AND
        DESTINATION=?
    CREATE_MESSAGE_TABLE = CREATE TABLE JMS_MESSAGES ( MESSAGEID INTEGER NOT NULL, DESTINATION VARCHAR(255)
        NOT NULL, TXID INTEGER, TXOP CHAR(1), MESSAGEBLOB BLOB, PRIMARY KEY (MESSAGEID, DESTINATION) )
    CREATE_TX_TABLE = CREATE TABLE JMS_TRANSACTIONS ( TXID INTEGER )
</attribute>

```

6.3.17 目的地 MBean

本节内容主要讨论 jbossmq-destinations-service.xml 和 jbossmq-service.xml 描述符中使用的目的地 MBean。

1. org.jboss.mq.server.jmx.Queue

JBossMQ 服务器使用 org.jboss.mq.server.jmx.Queue 定义 queue 目的地。其中, 该 MBean

的 JMX ObjectName 的 “name” 属性就是目的地名。比如，如果某 JMX MBean 开始于：

```
<mbean code="org.jboss.mq.server.jmx.Queue"
      name="jboss.mq.destination:service=Queue,name=testQueue">
```

则 JMX ObjectName 是 “jboss.mq.destination:service=Queue,name=testQueue”。queue 名为 “testQueue”。可配置的属性如下。

- **DestinationManager**: 服务器中目的地管理服务的 JMX ObjectName。开发者借助于 <depends optional-attribute-name="DestinationManager"> 标签能够设置该属性。
- **SecurityManager**: 用于验证客户请求的安全性管理器服务的 JMX ObjectName。开发者借助于 <depends optional-attribute-name="SecurityManager"> 标签能够设置该属性。
- **SecurityConf**: 该元素指定 XML 片段，其提供了访问控制列表。SecurityManager 使用该列表为客户提供目的地授权操作。其内容模型与图 6-2 给出的 Security Manager SecurityConf 属性一致。
- **JNDIName**: queue 对象绑定的 JNDI 位置。默认值为 “queue/queue-name”。
- **MaxDepth**: 存在于目的地的未处理消息上限。如果超出，则试图添加新消息，会抛出 org.jboss.mq.DestinationFullException。很多场合都可能超出 MaxDepth 值，比如当消息找到进入 queue 的窍门时。再比如，正处于提交阶段的事务在浏览 queue 的当前大小时，忽略了当前事务或其他事务可能添加的任何消息。这样做的原因是，当消息添加到物理 queue 时，开发者都不希望处于提交阶段的事务失败。
- **MessageCounterHistoryDayLimit**: 计算目的地消息的历史天数。如果小于 0，则不限制天数；如果等于 0，则不给出消息历史；如果大于 0，则计算该大小的历史天数。

其他用于提供统计信息的只读属性如下。

- **MessageCounter**: org.jboss.mq.sever.MessageCounter 实例数组，为目的地提供统计信息。
- **QueueDepth**: 等待消息的当前 backlog。
- **ReceiversCount**: 关联 queue 的当前接收者数量。
- **ScheduledMessageCount**: queue 中等待消息分发时间片的消息数量。
- **String listMessageCount()**: 该操作生成 HTML 表，包含如下内容。
 - **Type**: 表明目的地类型，queue 或 topic。
 - **Name**: 目的地名。
 - **Subscription**: 某 topic 的订阅 id。
 - **Durable**: boolean 标志位，表明某 topic 订阅是否是持久的。
 - **Count**: 分发到目的地的消息数量。
 - **CountDelta**: 自从上次访问 Count 以来，消息数量的变化。
 - **Depth**: 目的地中的消息数量。
 - **DepthDelta**: 自从上次访问 Depth 以来，目的地中消息数量的变化。
 - **Last Add**: 格式为 DataFormat.SHORT/DataFormat.MEDIUM 的日期/时间字

符串，表明最近一次添加消息到目的地的时间。

- **void resetMessageCounter():** 清零所有的目的地计数器和最近添加时间。
- **String listMessageCounterHistory():** 该操作显示 HTML 表，展示历史天数中每天每个小时的消息数量。
- **void resetMessageCounterHistory():** 该操作重置历史天数。

2. org.jboss.mq.server.jmx.Topic

JBossMQ 服务器使用 org.jboss.mq.server.jmx.Topic 定义 topic 目的地。其中，该 MBean 的 JMX ObjectName 的“name”属性就是目的地名。比如，如果某 JMX MBean 开始于：

```
<mbean code="org.jboss.mq.server.jmx.Topic"
name="jboss.mq.destination:service=Topic,name=testTopic">
```

则 JMX ObjectName 是“jboss.mq.destination:service=Topic,name=testTopic”。topic 名为“testTopic”。可配置的属性如下。

- **DestinationManager:** 服务器中目的地管理服务的 JMX ObjectName。开发者借助于<depends optional-attribute-name="DestinationManager">标签能够设置该属性。
- **SecurityManager:** 用于验证客户请求的安全性管理器服务的 JMX ObjectName。开发者借助于<depends optional-attribute-name="SecurityManager">标签能够设置该属性。
- **SecurityConf:** 该元素指定 XML 片段，其提供了访问控制列表。SecurityManager 使用该列表为客户提供目的地授权操作。其内容模型同图 6-2 给出的 SecurityManager SecurityConf 属性一致。
- **JNDIName:** topic 对象绑定的 JNDI 位置。默认值为“topic/topic-name”。
- **MaxDepth:** 存在于目的地的未处理消息上限。如果超出，则试图添加新消息，会抛出 org.jboss.mq.DestinationFullException。很多场合都可能超出 MaxDepth 值，比如当消息找到进入 topic 的窍门时。再比如，正处于提交阶段的事务在浏览 topic 的当前大小时，忽略了当前事务或其他事务可能添加的任何消息。这样做的原因是，当消息添加到物理 topic 时，大家都不希望处于提交阶段的事务失败。
- **MessageCounterHistoryDayLimit:** 计算目的地消息的历史天数。如果小于 0，则不限制天数；如果等于 0，则不给出消息历史；如果大于 0，则计算该大小的历史天数。

其他用于提供统计信息的只读属性如下。

- **AllMessageCount:** topic 中的所有消息数量。
- **AllSubscriptionsCount:** 持久和非持久订阅数量。
- **DurableMessageCount:** 持久订阅 topic 中的消息数量。
- **DurableSubscriptionsCount:** 持久订阅者的数量。
- **MessageCounter:** org.jboss.mq.sever.MessageCounter 实例数组，为目的地提供统计信息。
- **NonDurableMessageCount:** 非持久订阅 topic 中的消息数量。
- **NonDurableSubscriptionsCount:** 非持久订阅者的数量。

- **String listMessageCount():** 该操作生成 HTML 表, 包含如下内容。
 - **Type:** 表明目的地类型, queue 或 topic。
 - **Name:** 目的地名。
 - **Subscription:** 某 topic 的订阅 id。
 - **Durable:** boolean 标志位, 表明某 topic 订阅是否是持久的。
 - **Count:** 分发到目的地的消息数量。
 - **CountDelta:** 自从上次访问 Count 以来, 消息数量的变化。
 - **Depth:** 目的地中的消息数量。
 - **DepthDelta:** 自从上次访问 Depth 以来, 目的地中消息数量的变化。
 - **Last Add:** 格式为 DataFormat.SHORT/DataFormat.MEDIUM 的日期/时间字符串, 表明最近一次添加消息到目的地的时间。
- **void resetMessageCounter():** 清零所有的目的地计数器和最近添加时间。
- **String listMessageCounterHistory():** 该操作显示 HTML 表, 展示历史天数中每天每个小时的消息数量。
- **void resetMessageCounterHistory():** 该操作重置历史天数。

6.3.18 借助于 JMX 管理

开发者借助于 JMX 能够访问 JBossMQ 统计资料和若干个管理功能。借助于 Web 应用或借助于 JMX API 实现的应用, 开发者能够实现 JMX 交互功能。本书建议开发者使用 <http://localhost:8080/jmx-console> Web 应用来熟悉运行在服务器中的 JBossMQ JMX MBean。其中, 借助于 jmx-console Web 应用能够调用这些 MBean 的方法。本节只是给出管理员必须完成最常见的运行时的管理任务。

1. 运行时创建 queue

那些需要在运行时动态创建 queue 的应用程序, 开发者可以使用目的地管理器 MBean 的 createQueue 方法。

```
void createQueue(String name, String jndiLocation)
```

该方法创建的 queue 名为 name, 绑定的 JNDI 名为 jndiLocation。如果服务器重启, 借助于这种方式创建的 queue 将消失。为了销废以前创建的 queue, 可以使用如下方法。

```
void destroyQueue(String name)
```

2. 运行时创建 topic

那些需要在运行时动态创建 topic 的应用程序, 开发者可以使用目的地管理器 MBean 的 createTopic 方法。

```
void createTopic(String name, String jndiLocation)
```

该方法创建的 topic 名为 name, 绑定的 JNDI 名为 jndiLocation。如果服务器重启, 借助于这种方式创建的 topic 将消失。为了销废以前创建的 topic, 可以使用:

```
void destroyTopic(String name)
```

3. 运行时管理 JBossMQ 用户 id

开发者可以在运行时使用 `org.jboss.mq.sm.file.DynamicStateManager` MBean 添加和删除用户 id、角色。为添加用户 id，可以使用如下方法。

```
void addUser(String name, String password, String clientID)
```

该方法使用给定的 `name` 和 `password` 创建了用户 id，并将其 id 配置成特定的 `clientID`。为删除以前创建的用户 id，可以使用如下方法。

```
void removeUser(String name)
```

为管理属于某用户 id 的角色，开发者需要使用如下方法集合以创建角色、删除角色、将用户添加给角色及从角色中删除用户。

```
void addRole(String name)
void removeRole(String name)
void addUserToRole(String roleName, String user)
void removeUserFromRole(String roleName, String user)
```

6.4 指定 MDB JMS 供应商

至此，开发者都已经看到了标准的 JMS 客户端/服务器端架构。JMS 规范定义了高级接口，允许并发处理目的地中的消息，并将这些功能统称为应用服务器设施（Application Server Facility, ASF）。支持并发处理消息的两个接口是：`javax.jms.ServerSessionPool` 和 `javax.jms.ServerSession`。当需要并发处理消息时，应用服务器必须提供上述接口。因此，组成 JBossMQ ASF 的组件集合不仅涉及到 JBossMQ 组件，还有 JBoss 服务器组件。JBoss 服务器 MDB 容器利用 JMS 服务 ASF 并发处理发送给 MDB 的消息。

ASF 域的职责由 JMS 规范做了完美的定义，因此这里不讨论如何实现 ASF 组件。然而，本文打算讨论如何使用 MBean 集成 JBoss MDB 层使用的 ASF 组件，使得开发者能够替换应用服务器接口或 JMS 供应商接口实现。

接下来，以 `org.jboss.jms.jndi.JMSProviderLoader` MBean 为出发点，它负责装载 `org.jboss.jms.jndi.JMSProviderAdaptor` 接口实例到 JBoss 服务器中，并将其绑定到 JNDI。抽象的 `JMSProviderAdaptor` 接口定义了如何获得 JMS 供应商的 JNDI 根上下文。它为用于根 `InitialContext` 的 `Context.PROVIDER_URL` 定义了 `set` 和 `get` 方法，并且定义了 JNDI 根上下文中的 `QueueConnectionFactory` 和 `TopicConnectionFactory` 位置信息。这些都是引导 JMS 供应商所需的内容。通过将这些信息抽象到接口中，便能更换 JBoss MDB 容器中的 JMS ASF 供应商实现。`org.jboss.jms.jndi.JBossMQProvider` 是 `JMSProviderAdaptor` 接口实现，它为 JBossMQ JMS 供应商提供适配器。为用其他 JMS ASF 实现替换 JBossMQ 供应商，开发者只需要实现 `JMSProviderAdaptor` 接口，并且用该实现类名配置 `JMSProviderLoader`。本文后面将给出实例。

除了能替换 MDB 的 JMS 供应商外, 开发者还能够替换 javax.jms.ServerSessionPool 接口实现。方法如下: 使用 org.jboss.jms.asf.ServerSessionPoolLoader MBean 的 PoolFactoryClass 属性配置 org.jboss.jms.asf.ServerSessionPoolFactory 实现类名。其中, org.jboss.jms.asf.StdServerSessionPoolFactory 类是 ServerSessionPoolFactory 的默认工厂实现。

6.4.1 org.jboss.jms.jndi.JMSProviderLoader MBean

JMSProviderLoader MBean 服务创建的 JMS 供应商适配器被绑定到 JNDI 中。JMS 供应商适配器类实现了 org.jboss.jms.jndi.JMSProviderAdaptor 接口。MDB 容器需要使用它以独立于具体供应商的方式访问 JMS 服务供应商。JMSProviderLoader 的可配置属性如下。

- **ProviderName:** JMS 供应商的惟一名。JBoss 将使用它绑定 JMSProviderAdaptor 实例到 JNDI “java:/<ProviderName>”, 除非 AdapterJNDIName 属性覆盖了它。
- **ProviderAdaptorClass:** 用于创建 org.jboss.jms.jndi.JMSProviderAdaptor 接口实现的全限定类名。如果更换 JMS 供应商, 比如 SonicMQ, 则开发者必须实现 JMSProviderAdaptor 接口, 从而能够管理 InitialContext 的供应商 URL 及 JNDI 中 QueueConnectionFactory 和 TopicConnectionFactory 的位置信息。
- **AdaptorJNDIName:** 指定绑定到 JMSProviderAdaptor 实例的 JNDI 名。
- **ProviderURL:** JNDI Context.PROVIDER_URL 值, 创建 JMS 供应商根 InitialContext 时需要使用它。
- **QueueFactoryRef:** 供应商 javax.jms.QueueConnectionFactory 绑定的 JNDI 名。
- **TopicFactoryRef:** 供应商 javax.jms.TopicConnectionFactory 绑定的 JNDI 名。

列表 6-14 给出了访问远程 JBossMQ 服务器的 JMSProviderLoader。

列表 6-14 访问远程 JBossMQ 服务器的 JMSProviderLoader

```
<mbean code="org.jboss.jms.jndi.JMSProviderLoader"
  name="jboss.mq:service=JMSProviderLoader,name=RemoteJBossMQProvider"
  >
  <attribute name="ProviderName">RemoteJMSProvider</attribute>
  <attribute name="ProviderUrl">jnp://remotehost:1099</attribute>
  <attribute name="ProviderAdaptorClass">
    org.jboss.jms.jndi.JBossMQProvider
  </attribute>
  <attribute name="QueueFactoryRef">XAConnectionFactory</attribute>
  <attribute name="TopicFactoryRef">XAConnectionFactory</attribute>
</mbean>
```

列表 6-15 给出了 jboss.xml 片段, 其中该列表通过 MDB Invoker 配置引用了 RemoteJMSProvider。

列表 6-15 指定 MDB JMS 供应商适配器的 jboss.xml 片段

```
...
<proxy-factory-config>
  <JMSProviderAdapterJNDI>RemoteJMSProvider</JMSProviderAdapterJNDI>
```



```

<ServerSessionPoolFactoryJNDI>StdJMSPool
</ServerSessionPoolFactoryJNDI>
<MaximumSize>15</MaximumSize>
<MaxMessages>1</MaxMessages>
<MDBConfig>
  <ReconnectIntervalSec>10</ReconnectIntervalSec>
  <DLQConfig>
    <DestinationQueue>queue/DLQ</DestinationQueue>
    <MaxTimesRedelivered>10</MaxTimesRedelivered>
    <TimeToLive>0</TimeToLive>
  </DLQConfig>
</MDBConfig>
</proxy-factory-config>

```

因为开发者可能指定多个<invoker-proxy-binding>，所以通过配置不同 JMSProvider AdapterJNDI 设置的多个绑定，能够实现 MDB 监听多个服务器上的相同 queue/topic。

另外，开发者可以使用 JCA 配置集成 JMS 供应商，如列表 6-16 所示。

列表 6-16 借助于 JCA 集成 JMS 供应商适配器的 jms-ds.xml 描述符

```

<tx-connection-factory>
  <jndi-name>RemoteJmsXA</jndi-name>
  <xa-transaction/>
  <adapter-display-name>JMS Adapter</adapter-display-name>
  <config-property name="JMSProviderAdapterJNDI"
    type="java.lang.String">RemoteJMSProvider</config-property>
  <config-property name="SessionDefaultType"
    type="java.lang.String">javax.jms.Topic</config-property>

  <security-domain-and-application>JmsXARealm
  </security-domain-and-application>
</tx-connection-factory>

```

6.4.2 org.jboss.jms.asf.ServerSessionPoolLoader MBean

ServerSessionPoolLoader MBean 服务管理工厂，即能够创建 MDB 容器使用的 javax.jms.ServerSessionPool 对象。其可配置属性如下。

- **PoolName**: 会话池的惟一名。使用它将 ServerSessionPoolFactory 实例绑定到 JNDI, “java:/PoolName”。
- **PoolFactoryClass**: org.jboss.jms.asf.ServerSessionPoolFactory 接口实现的全限定类名。
- **XidFactory**: MBean 服务的 JMX ObjectName。其中，当不需要两阶段提交时（即本地事务），使用该 MBean 服务生成 javax.transaction.xa.Xid 值。该 MBean 服务必须提供一能够返回 org.jboss.tm.XidFactoryMBean 实例的操作。

6.4.3 集成其他 JMS 供应商

大家已经知道，开发者能够使用外部实现替换 JBossMQ JMS 实现。如下给出各种可能的替换方法。

- 替换 `JMSProviderLoader` 类。该类必须能够实例化正确的 JNDI 上下文，以用于外部 JMS 供应商管理对象的通信。
- 使用 `ExternalContext MBean` 将外部 JMS 供应商管理对象集成到 JBoss JNDI 树中。
- 使用 `MBean` 实例化外部 JMS 对象到 JBoss JNDI 树中。用于 WebSphere MQ 的相关实例可以通过 http://sourceforge.net/tracker/index.php?func=etail&aid=753022&group_id=22866&atid=376687 找到。

第 7 章 JBoss 之连接器——JCA

配置和架构

本章讨论 JBoss 服务器实现的 J2EE 连接器架构 (J2EE Connector Architecture, JCA)。JCA 是资源管理器集成 API, 其目标是能够类似于标准 JDBC API 访问关系型数据, 以标准化方式访问非关系型资源。本章先介绍 JCA API 的使用, 然后介绍 JBoss 3.2.x 中的 JCA 架构。

7.1 JCA 概述

J2EE 1.3 包含的连接器架构规范能够将事务性、安全的资源适配器集成到 J2EE 应用服务器环境中。JCA 主页 (<http://java.sun.com/j2ee/connector>) 提供完整的 JCA 规范, 可供开发者下载使用。JCA 规范描述资源管理器, 比如企业信息系统 (Enterprise Information System, EIS)。EIS 系统实例有企业资源计划、大型主机事务处理、非 Java 遗留应用等等。

专注于 EIS 的主要原因是, 企业软件系统要求具有良好的事务性、安全性及扩展性。然而, JCA 适用于以安全、可扩展及事务的方式集成到 JBoss 中的任何资源。这里所介绍的资源适配器是通用概念, 没有针对具体的 EIS 环境。

连接器架构定义了服务供应商接口 (Service Provider Interface, SPI), 用于集成应用服务器和资源管理器的事务、安全性及连接管理服务。SPI 定义了资源适配器和应用服务器之间的系统级契约。

连接器架构也定义了访问资源的公共客户接口 (Common Client Interface, CCI), 用于 EIS 开发工具和其他集成资源的高级用户。CCI 为这类工具尽可能不提供具体 EIS 相关代码。通常情况下, J2EE 开发者使用工具或资源接口访问资源, 而不是直接使用 CCI。原因在于 CCI 不是具体类型的 API。为能够有效地使用它, 必须和元数据协同使用。其中, 这里的元数据指, 那些描述如何将通用 CCI API 映射成资源管理器内部使用的具体资源管理器数据类型的数据。

连接器架构的目的在于使资源厂商能够为它的产品提供标准的适配器。资源适配器是系统级软件驱动程序, Java 应用使用它连接到资源。资源适配器插入到应用服务器中, 从而为资源管理器、应用服务器及企业应用之间提供连接。资源厂商只需要实现一次 JCA 兼容适配器, 便能够在具有 JCA 能力的应用服务器中使用资源管理器。

应用服务器厂商只需扩展其架构一次, 以支持连接器架构, 然后就能够保证它与多个资源管理器的无缝集成。同样, 资源管理器厂商提供标准资源适配器, 并插入到支持 JCA 的应用服务器中。

图 7-1 描述了扩展的应用服务器能够为 JCA SPI 提供支持, 从而允许资源适配器集成服务器的连接池、事务管理及安全性管理设施服务。定义系统契约的集成 API 包括如下部

分：

- 连接管理：契约，允许应用服务器生成资源连接池。引入池管理能够提供扩展能力。通常情况下，创建资源连接相当消耗系统资源，因此将创建的对象放入池中，能够实现资源的更有效重用和管理。
- 事务管理：契约，允许应用服务器中的事务管理器管理资源管理器参与的事务。
- 安全性管理：安全访问资源管理器的契约。

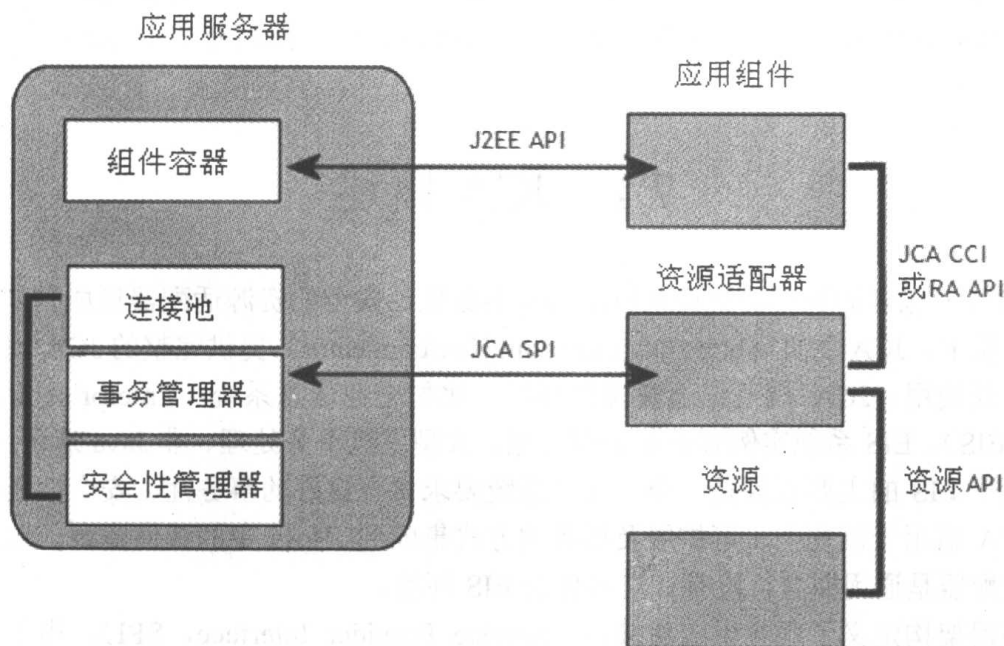


图 7-1 J2EE 应用服务器与 JCA 资源适配器的关系

资源适配器实现了系统契约的资源管理器部分，它必须使用应用服务器连接池，提供事务资源信息和使用安全集成信息。资源适配器也将资源管理器暴露给应用服务器组件，使用 CCI 和（或）具体资源适配器 API 能够实现上述目的。

遵循组件约定的应用组件使用标准 J2EE 容器集成到应用服务器中。对于 EJB 组件而言，该约定由 EJB 规范定义。应用组件同资源适配器交互的方式与它和其他标准资源工厂相同，比如 `javax.sql.DataSource` JDBC 资源工厂。其与 JCA 资源适配器的惟一差别在于，如果资源适配器支持独立的 CCI API，则客户可以使用这些 CCI API。

JCA 1.0 规范阐述了 JCA 架构参与者之间的关系，其中包括 JCA SPI、CCI 及 JTA 包。它们之间的关系如图 7-2 所示。

JBossCX 架构提供了应用服务器具体类的实现。图 7-2 给出了两接口的实现：`javax.resource.spi.ConnectionManager` 和 `javax.resource.spi.ConnectionEventListener` 接口。接下来讨论的 JBossCX 架构中将谈到这些实现的重要细节。

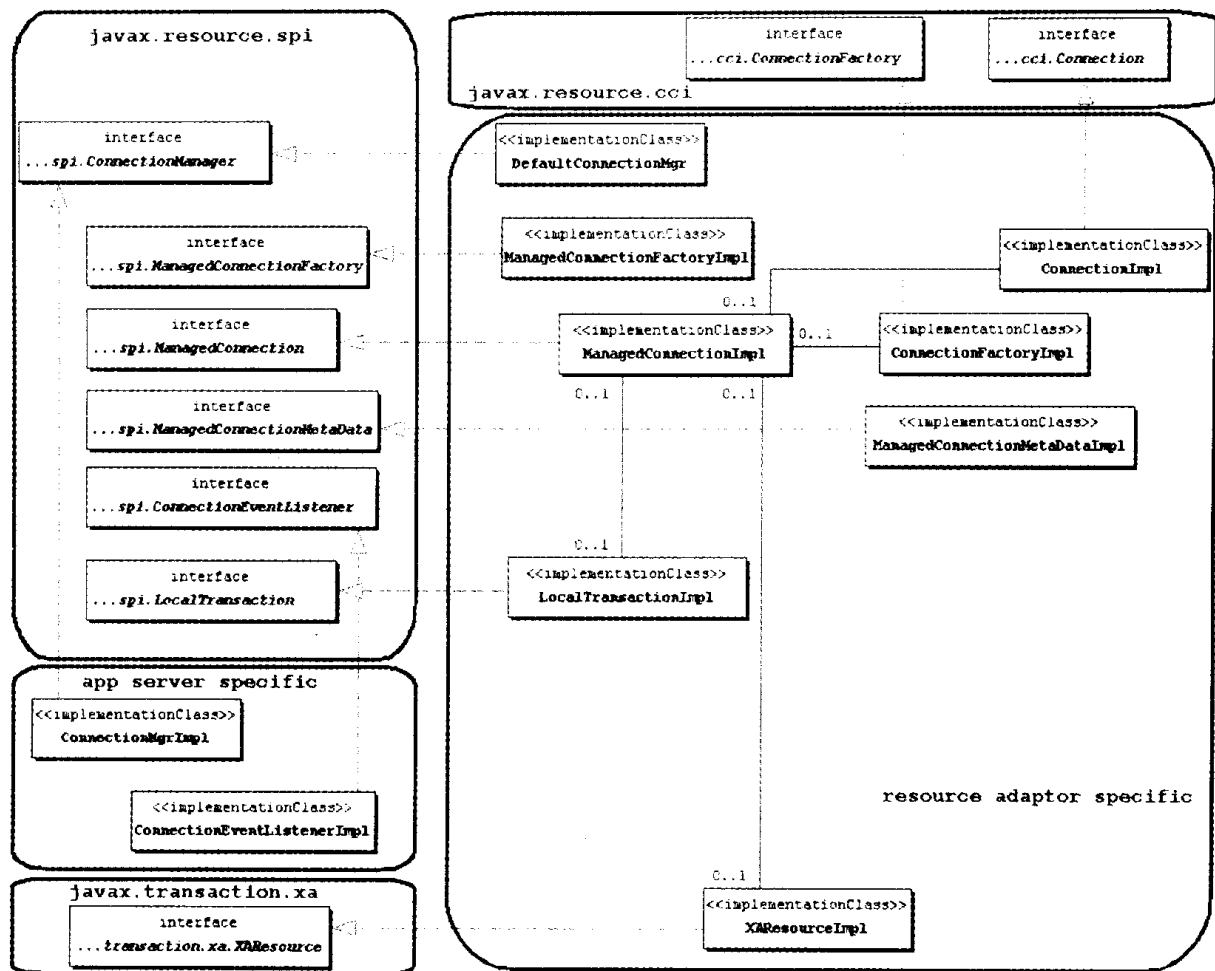


图 7-2 JCA 1.0 规范中用于连接管理架构的类图

7.2 JBossCX 架构概述

JBossCX 框架为使用 JCA 资源适配器提供了所需的应用服务器架构扩展。它主要包括：连接池和管理扩展，以及用于装载资源适配器到 JBoss 服务器的若干 MBean。图 7-3 展开了图 7-2 的具体视图，并描述了 JBoss JCA 层是如何实现应用服务器具体扩展的。该图还给出了后续将要研究的文件系统资源适配器实例。

RAR 部署单元是由 3 个耦合的 MBean 构成，分别是：

- org.jboss.resource.RARDeployment
- org.jboss.resource.connectionmanager.RARDeployment
- org.jboss.resource.connectionmanager.BaseConnectionManager2

其中，org.jboss.resource.RARDeployment 只是简单地封装了 RAR META-INF/ra.xml 描述符元数据，并将这些信息简单地表现为 DynamicMBean，从而提供给 org.jboss.resource.connectionmanager.RARDeployment MBean。

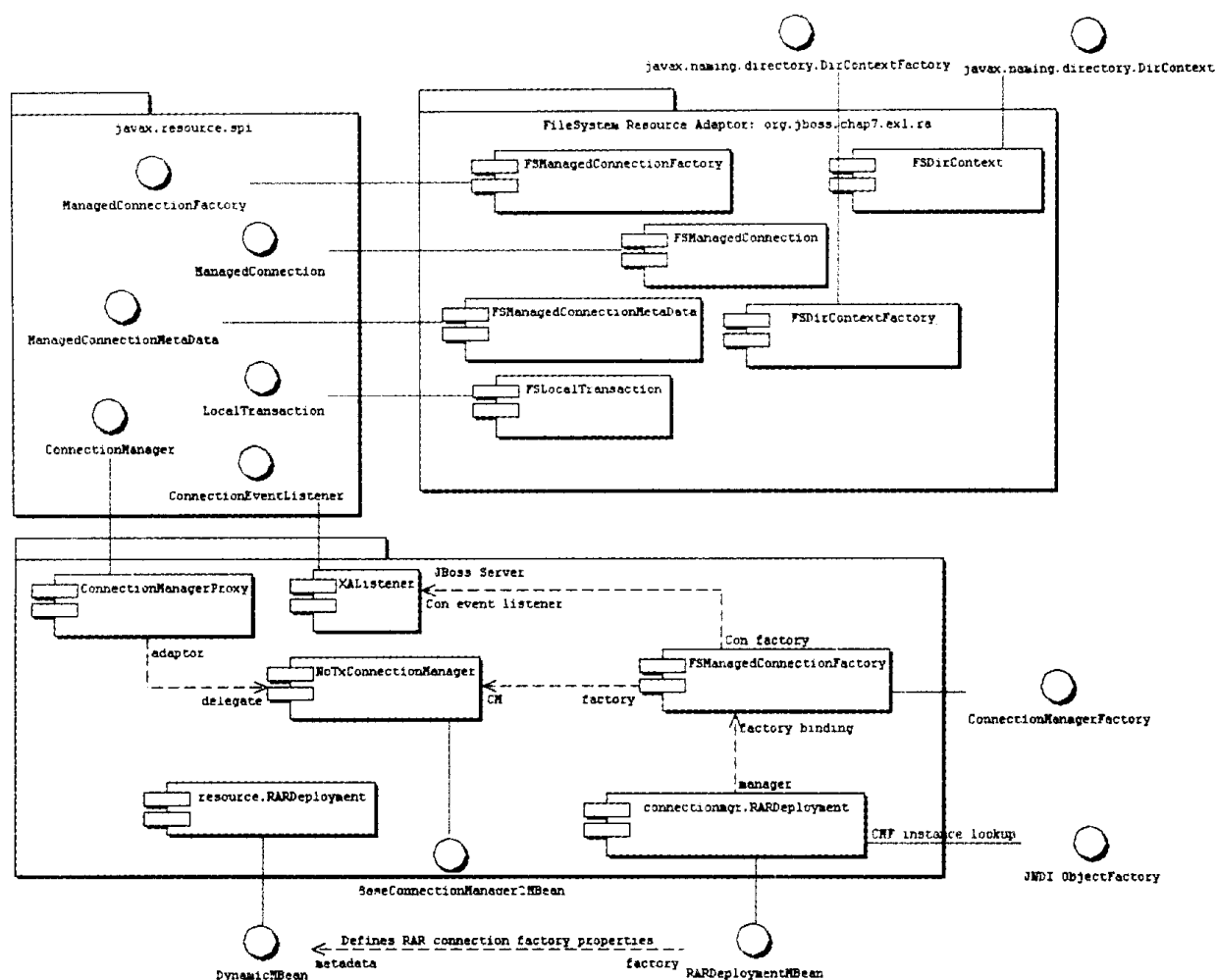


图 7-3 JBoss JCA 实现组件

RARDeployer 服务处理含有资源适配器的存档文件（RAR）的部署。当部署 RAR 文件时，将创建 `org.jboss.resource.RARDeployment` MBean。为实现应用组件能够使用资源适配器，必须首先部署 RAR 文件。对于各个已部署的 RAR 文件，必须配置和绑定一个或多个连接工厂到 JNDI 中，使用 JBoss 服务描述符能完成该任务，即配置 `org.jboss.resource.connectionmanager.BaseConnectionManager2` MBean 实现。当然，还包括其依赖的 `org.jboss.resource.connectionmanager.RARDeployment`。

7.2.1 BaseConnectionManager2 MBean

`org.jboss.resource.connectionmanager.BaseConnectionManager2` MBean 是 JCA 规范要求的、用于各种不同连接管理器类型的基类。包括的子类如下：

- `org.jboss.resource.connectionmanager.NoTxConnectionManager`
- `org.jboss.resource.connectionmanager.LocalTxConnectionManager`
- `org.jboss.resource.connectionmanager.XATxConnectionManager`

它们对应的资源适配器分别是：不支持事务、支持本地事务、支持 XA 事务。开发者需要根据欲使用的事务语义和 JCA 资源适配器支持的事务能力来选择相应的基类。

BaseConnectionManager2 MBean 支持的常见属性如下:

- **ManagedConnectionFactoryName:** ManagedConnectionFactoryName 属性指定创建 javax.resource.spi.ManagedConnectionFactory 实例的 MBean 的 ObjectName。为配置 ManagedConnectionFactoryName 属性值,开发者一般都需要借助于 depends 元素,将 org.jboss.resource.connectionmanager.RARDeployment MBean 配置成嵌入式 MBean,而不是单独引用该 MBean。该 MBean 必须提供具有如下方法定义的操作:

```
javax.resource.spi.ManagedConnectionFactory startManagedConnectionFactory (javax.resource.spi.  
ConnectionManager)
```

- **ManagedConnectionPool:** ManagedConnectionPool 属性指定,用于连接管理器的连接池 MBean 的 ObjectName。该 MBean 应该提供 ManagedConnectionPool 属性,以定义 org.jboss.resource.connectionmanager.ManagedConnectionPool 实现。通常使用 depends 元素将其嵌入在 MBean 中,而不是使用 ObjectName 引用现有 MBean。默认值,org.jboss.resource.connectionmanager.JBossManagedConnectionPool。相应的可配置属性后续还有阐述。
- **CachedConnectionManager:** CachedConnectionManager 属性指定连接管理器使用的 org.jboss.resource.connectionmanager.CachedConnectionManager MBean 服务的 ObjectName。通常使用 depends 元素指定服务器中惟一 CachedConnectionManager MBean 的 ObjectName。其中,标准的 depends 元素值为“jboss.jca:service=CachedConnectionManager”。
- **SecurityDomainJndiName:** SecurityDomainJndiName 属性指定安全性域的 JNDI 名,以用于资源连接的认证和授权服务。一般情况下,它是以“java:jaas/<domain>”形式给出。其中,<domain>值指 JAAS 登录模块配置文件,即 conf/login-config.xml 的入口名,它定义具体使用哪个 JAAS 登录模块完成认证过程。安全性设置更多内容,请参考第8章内容。
- **JaasSecurityManagerService:** JaasSecurityManagerService 属性指定安全性管理器服务的 ObjectName。它是 conf/jboss-service.xml 描述符中定义的安全性管理器的 MBean 名。当前,该属性取值为“jboss.security:service=JaasSecurityManager”。JBoss 的后续版本可能会删除该属性。

7.2.2 RARDeployment MBean

org.jboss.resource.connectionmanager.RARDeployment MBean 管理 ManagedConnectionFactory 的配置和实例化工作。其具体过程是使用 RAR 描述符 META-INF/ra.xml 中的资源适配器元数据和 RARDeployment 属性完成的。可配置属性如下:

- **OldRarDeployment:** OldRarDeployment 属性指定包含资源适配器元数据的 org.jboss.resource.RARDeployment MBean 的 ObjectName。其命名形式为:jboss.jca:service=RARDeployment,name=<ra-display-name>。其中,<ra-display-name>指

ra.xml 描述符中 display-name 属性值。当部署 RAR 文件时, RARDeployer 会创建它。JBoss 的后续版本可能会删除该属性。

- **ManagedConnectionFactoryProperties**: ManagedConnectionFactoryProperties 属性指定(name,type,value)集合, 即用于定义 ManagedConnectionFactory 实例的属性。因此, 属性名依赖于资源适配器, 即 ManagedConnectionFactory 实例。该属性的内容结构如下:

```
<properties>
  <config-property>
    <config-property-name>Attr0Name</config-property-name>
    <config-property-type>Attr0Type</config-property-type>
    <config-property-value>Attr0Value</config-property-value>
  </config-property>
  <config-property>
    <config-property-name>Attr1Name</config-property-name>
    <config-property-type>Attr2Type</config-property-type>
    <config-property-value>Attr2Value</config-property-value>
  </config-property>
  ...
</properties>
```

其中, AttrXName 是第 X 个属性名, AttrXType 是第 X 个属性的全限定 Java 类型, AttrXValue 用字符串表示其值。使用 java.beans.PropertyEditor 类能够完成 String 到 AttrXType 类型的转换。

- **JndiName**: JNDI 名。资源适配器客户使用该 JNDI 名, 以获得资源适配器本身的连接工厂。客户使用该 JndiName 能够获得 javax.resource.cci.ConnectionFactory, 或者具体资源适配器连接工厂。“java:<JndiName>”是完整的 JNDI 名, 其中往 JndiName 属性值前添加了前缀“java:/”, 从而防止在 JBoss 服务器 JVM 外部使用该连接工厂。JBoss 的后续版本可能会将该限制实现成可配置的。

7.2.3 JBossManagedConnectionPool MBean

org.jboss.resource.connectionmanager.JBossManagedConnectionPool MBean 是连接池 MBean。通常都是把 JBossManagedConnectionPool 作为嵌入式 MBean, 即 BaseConnectionManager2 ManagedConnectionPool 属性的取值。当配置连接管理器 MBean 时, 开发者通常把它嵌入在连接管理器描述符中的池配置中。可配置属性如下:

- **MinSize**: MinSize 属性指定池中应该持有的最少连接数量。如果连接请求的 Subject 未知, 则不会创建这些连接。每个子池都将创建 MinSize 个连接。
- **MaxSize**: MaxSize 属性指定池中允许持有的最大连接数量。每个子池创建的连接数不会超过 MaxSize。
- **BlockingTimeoutMillis**: BlockingTimeoutMillis 属性指定, 用于等待连接, 但还未抛出异常所能阻塞的最大时间。请注意, 这里的阻塞只是等待连接许可。如果创建新连接消耗了大量的时间, 也不会抛出异常。

- **IdleTimeoutMinutes:** IdleTimeoutMinutes 属性指定, 连接在关闭前能空闲的最大时间。其中, 实际的最大时间还依赖于空闲 IdleRemover 线程的扫描时间, 该扫描时间是所有池中最小空闲超时的 1/2。
- **NoTxSeperatePools:** 标志位。如果为 true, 则加倍可用的池。一个池用于事务外连接使用, 另一个池用于事务中连接使用。实际上, 如果没有客户请求, 则不会创建这些池。只有设置 LocalTxConnectionManager 和 XATxConnectionManager 的池参数时, 才会使用到该属性。比如, 对于 Oracle (也可能存在其他厂商) XA 实现, 开发者可以通过或不通过 JTA 事务使用 XA 连接。
- **Criteria:** Criteria 属性指定如下内容, 即使用连接关联的安全性域的 JAAS javax.security.auth.Subject, 还是使用应用提供的参数 (比如, 来自 getConnection(user, ps)) 来区分池中的连接。该属性的取值范围如下:
 - ByContainerAndApplication (两者都使用)
 - ByContainer (使用 Subject)
 - ByApplication (仅仅使用应用提供的参数)
 - ByNothing (如果适配器支持重新认证, 则所有连接都是等效的)

7.2.4 CachedConnectionManager MBean

org.jboss.resource.connectionmanager.CachedConnectionManager MBean 服务不仅管理元敏感 (meta-aware) 对象 (那些通过拦截器栈访问) 和连接 handle 之间的关系, 而且还管理用户事务和连接 handle 之间的关系。通常, 只存在单个这样的 MBean, 并配置在核心 jboss-service.xml 描述符中。所有的 BaseConnectionManager2 实例, org.jboss.resource.connectionmanager.CachedConnectionInterceptor, 以及 JTA javax.transaction.UserTransaction 实现都会使用该 MBean。其可配置属性如下:

- **SpecCompliant:** boolean 标志位, 用于兼容 JCA 规范中 “nonshareable” 连接的重连接处理。它配置如下内容, 即是否允许在某访问操作中使用其他访问操作打开的连接。请注意, 如果使用这种特征, 则连接关闭处理失效。
- **Debug:** boolean 属性, 用于连接关闭处理。当 EJB 方法调用完成时, 事务同步注册那些未关闭的连接。如果事务结束时没有关闭连接, 则有错误抛出, 然后 JBoss 会关闭这些连接。该特性只适用于开发目的, 为优化产品运行的性能, 在通常情况下请关闭它。
- **TransactionManagerServiceName:** TransactionManagerServiceName 属性指定 JTA 事务管理器服务的 JMX ObjectName。事务管理器将同步连接关闭处理, 该属性指定待用的事务管理器。

7.2.5 JCA 资源适配器实例纲要

为总结上文讨论的 JBoss JCA 框架, 本文创建并部署了单个非事务性的资源适配器。这里只是简单地给出纲要 (skeleton) 实现, 即实现要求的接口并记录所有的方法调用。本节不讨论资源适配器厂商的具体需求, JCA 规范中有详细论述。本适配器的用意在于演

示为 JBoss 创建、部署 RAR 的步骤。另外,还包括适配器与 JBoss 的具体交互方式。

开发者可以将这里创建的适配器作为非事务性文件系统适配器的切入点。适配器实例源代码位于 `src/main/org/jboss/chap7/ex1` 目录下。如图 7-4 所示的类图,它显示 `javax.resource.spi` 要求接口到资源适配器实现的映射。

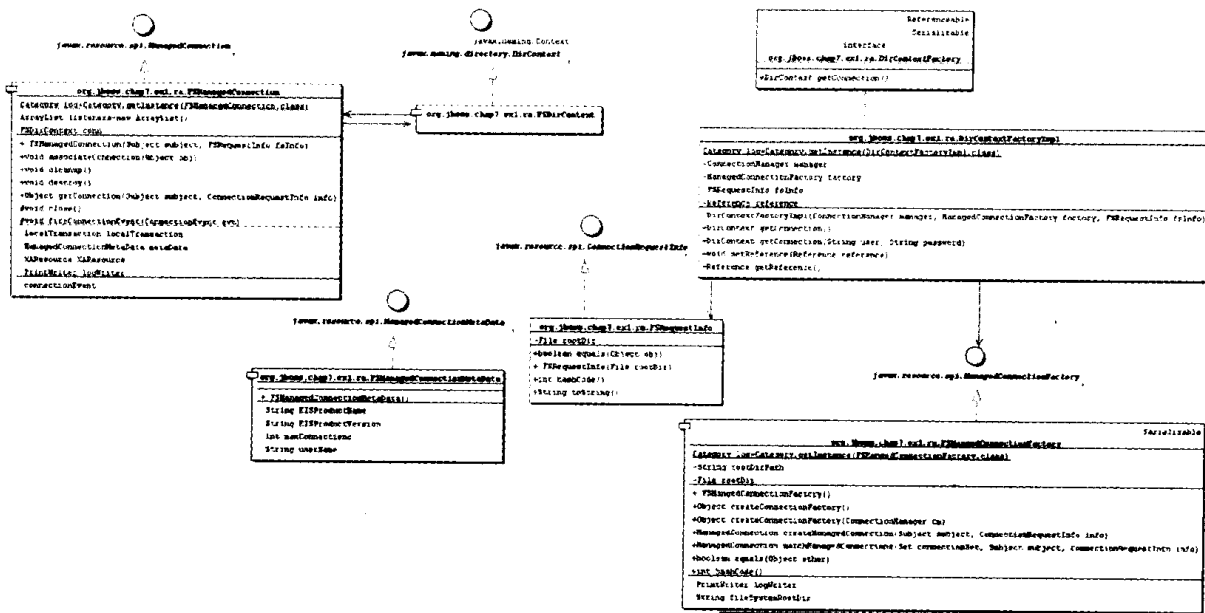


图 7-4 文件系统 RAR 类图

接下来,开发者需要编译该适配器,并将它部署到 JBoss 服务器上。然后,运行客户端实例以调用使用了资源适配器的 EJB,从而能够看出完成整个过程所需的步骤。最后,开发者还可以浏览 JBoss 服务器日志,并且能够查看到 JBoss JCA 框架与资源适配器的交互过程,使得开发者能够更好地理解 JCA 系统级契约中的组件。

在光盘目录下,请开发者执行如下 ant 命令行,从而完成资源适配器的编译工作。

```
[nr@toki]$ ant -Dchap=chap7 build-chap
Buildfile: build.xml

...
prepare:
[mkdir] Created dir: /Users/orb/Desktop/jboss/docs323/examples/output/chap7

chap7-ex1-rar:
[jar] Building jar: /Users/orb/Desktop/jboss/docs323/examples/output/chap7/ra.jar
[jar] Building jar: /Users/orb/Desktop/jboss/docs323/examples/output/chap7/chap7-ex1.rar

prepare:

chap7-ex1-jar:
[jar] Building jar: /Users/orb/Desktop/jboss/docs323/examples/output/chap7/chap7-ex1.jar
BUILD SUCCESSFUL
```

Total time: 6 seconds

部署文件包括 chap7-ex1.sar 和 notxfs-service.xml 服务描述符。列表 7-1 给出了资源适配器实例的部署描述符，列表 7-2 给出了连接管理器 MBean 的服务描述符。

列表 7-1 非事务性文件系统资源适配器的部署描述符

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE connector PUBLIC
    "-//Sun Microsystems, Inc.//DTD Connector 1.0//EN"
    "http://java.sun.com/dtd/connector_1_0.dtd">
<connector>
    <display-name>File System Adapter</display-name>
    <vendor-name>JBoss Group</vendor-name>
    <spec-version>1.0</spec-version>
    <version>1.0</version>
    <eis-type>FileSystem</eis-type>
    <license>
        <description>IGPL</description>
        <license-required>>false</license-required>
    </license>
    <resourceadapter>
        <managedconnectionfactory-class>
            org.jboss.chap7.ex1.ra.FSMangedConnectionFactory
        </managedconnectionfactory-class>
        <connectionfactoryinterface>
            org.jboss.chap7.ex1.ra.DirContextFactory
        </connectionfactory-interface>
        <connectionfactory-implclass>
            org.jboss.chap7.ex1.ra.DirContextFactoryImpl
        </connectionfactory-impl-class>
        <connection-interface>javax.naming.directory.DirContext
        </connection-interface>
        <connection-impl-class>org.jboss.chap7.ex1.ra.FSDirContext
        </connection-impl-class>
        <transaction-support>NoTransaction</transaction-support>
        <config-property>
            <config-property-name>FileSystemRootDir</config-property-name>
            <config-property-type>java.lang.String</config-property-type>
            <config-property-value>/tmp/db/fs_store</config-propertyvalue>
        </config-property>
        <config-property>
            <config-property-name>UserName</config-property-name>
            <config-property-type>java.lang.String</config-property-type>
            <config-property-value></config-property-value>
        </config-property>
    </resourceadapter>
</connector>
```

```

    <config-property-name>Password</config-property-name>
    <config-property-type>java.lang.String</config-property-type>
    <config-property-value></config-property-value>
  </config-property>
  <authentication-mechanism>
    <authentication-mechanism-type>BasicPassword</authentication-mechanism-type>
    <credential-interface>javax.resource.security.PasswordCredential</credential-interface>
  </authentication-mechanism>
  <reauthentication-support>true</reauthentication-support>
  <security-permission>
    <description>Read/Write access is required to the contents of
    the FileSystemRootDir</description>
    <security-permission-spec>permission java.io.FilePermission
      "/tmp/db/fs_store/*", "read,write";</security-permission-spec>
  </security-permission>
</resource-adapter>
</connector>

```

列表 7-2 资源适配器 MBean 服务描述符 notxfds.xml

```

<!-- The non-transaction FileSystem resource adaptor service
configuration
-->
<connection-factories>
  <no-tx-connection-factory>
    <jndi-name>NoTransFS</jndi-name>
    <adapter-display-name>File System Adaptor</adapter-display-name>
    <config-property name="FileSystemRootDir"
      type="java.lang.String">/tmp/db/fs_store</config-property>
  </no-tx-connection-factory>
</connection-factories>

```

资源适配器部署描述符中的关键部分使用了粗体高亮显示。它们定义了资源适配器类，包含的元素如下：

- **display-name**：本章在讨论连接管理器工厂 MBean 时提到，通过 name 找到的 RARDeployment DynamicMBean 能够关联工厂和资源适配器类，而这个 name 就是根据 ra.xml 描述符中的 display-name 值获得的。这里的 name 是“File System Adaptor”，在连接管理器服务描述符中要使用它。
- **managedconnectionfactory-class**：managedconnectionfactory-class 元素给出了 javax.resource.spi.ManagedConnectionFactory 接口实现，即 org.jboss.chap7.ex1.ra.FSManagedConnectionFactory。
- **connectionfactory-interface**：connectionfactory-interface 元素指定，当客户查找 JNDI 中的连接工厂实例时，它将获得该接口。这里是适配器专属值，即 org.jboss.chap7.ex1.ra.DirContextFactory。
- **connectionfactory-impl-class**：connectionfactory-impl-class 元素指定，实现了

connectionfactory-interface 元素指定的类，即 org.jboss.chap7.ex1.ra.DirContextFactoryImpl。

- **connection-interface**: connection-interface 元素指定资源适配器连接工厂返回的接口，即 JNDI javax.naming.directory.DirContext。
- **connection-impl-class**: connection-impl-class 元素指定实现了 connection-interface 元素指定的类，即 org.jboss.chap7.ex1.ra.FSDirContext。
- **transaction-support**: transaction-support 元素指定资源适配器支持的事务级别。这里指 NoTransaction，即文件系统资源适配器并没有实现事务性工作。

ra.xml 描述符元素的完整解释，请参考 JCA 1.0 规范，或参考由 Sharma、Stearns 及 Ng 共同写作的《J2EE 连接器体系与企业应用集成》（《J2EE Connector Architecture and Enterprise Application Integration》）¹一书。

RAR 类和部署描述符只完成了资源适配器的定义，如果开发者需要使用资源适配器，还需将它集成到 JBoss 应用服务器中。前面已经讨论过通过连接工厂 MBean 能够完成集成过程。自从 JBoss 3.2 开始，服务器就提供了用于配置应用服务器 JCA 服务的简化描述符格式。“7.3.2 配置常见 JCA 适配器”节有讨论。列表 7-2 给出了 notxfs-ds.xml 描述符，开发者应该注意的细节有：

- jndi-name 元素：用于指定连接工厂绑定的 JNDI 名，该部署的绑定名为“java:/NoTransFS”。
- adapter-display-name 元素：同 ra.xml display-name 元素的取值。通过这两个取值，JCA 层能够知道如何关联连接工厂配置中的具体 RAR。
- config-property 元素：指定 ManagedConnectionFactoryProperties，从而为资源适配器连接工厂提供自定义配置。其中，这里 FileSystemRootDir 的类型是“java.lang.String”，元素值为“/tmp/db/fs_store”。

运行如下命令行，能够部署 RAR 和连接管理器配置到 JBoss 服务器中。

```
[nr@toki examples]$ ant -Dchap=chap7 config
Buildfile: build.xml
...
config:
  [copy] Copying 1 file to /tmp/jboss-3.2.3/server/default/deploy
  [copy] Copying 1 file to /tmp/jboss-3.2.3/server/default/deploy

BUILD SUCCESSFUL
Total time: 1 second
```

服务器控制台将显示如下信息：

```
01:11:30,945 INFO [MainDeployer] Starting deployment of package: file:/private/tmp/jboss-3.2.3/
server/default/deploy/chap7-ex1.rar
01:11:31,374 INFO [RARMetaData] License terms present. See deployment descriptor.
01:11:31,448 INFO [RARDeployer] nested deployment: file:/private/tmp/jboss-3.2.3/server/
```

¹ 译者注：该书中文版已经由电子工业出版社出版发行

```
default/tmp/deploy/tmp35575chap7-ex1.rar-contents/ra.jar
01:11:31,496 INFO [MainDeployer] Deployed package: file:/private/tmp/jboss-3.2.3/
server/default/deploy/chap7-ex1.rar
01:11:31,502 INFO [MainDeployer] Starting deployment of package: file:/private/tmp/jboss-3.2.3/
server/default/deploy/notxfs-ds.xml
01:11:31,783 INFO [RARDeployment] Started jboss.jca:service=ManagedConnection
Factory,name=NoTransFS
01:11:31,792 INFO [JBossManagedConnectionPool] Started jboss.jca:service=ManagedConnection
Pool,name=NoTransFS
01:11:31,848 INFO [NoTransFS] Bound connection factory for resource adapter for
ConnectionManager 'jboss.jca:service=NoTxCM,name=NoTransFS to JNDI name 'java:/NoTransFS'
01:11:31,852 INFO [NoTxConnectionManager] Started jboss.jca:service=
NoTxCM,name=NoTransFS
01:11:31,862 INFO [MainDeployer] Deployed package: file:/private/tmp/jboss-3.2.3/server/
default/deploy/notxfs-ds.xml
```

这些日志信息表明，ant 命令已成功部署资源适配器，并将连接工厂绑定到 JNDI “java:/NoTransFS”。其中，开发者需要忽略部署过程中出现的警告消息。由于上述 RARDeployment MBean 没有实现任何 Service 方法，它只是标准 MBean，而没有用做 JBoss 服务，因此会出现这些警告信息。

至此，本文打算用 J2EE 组件测试访问资源适配器。为此创建了一个试验性的无状态会话 Bean，具有单个方法 echo。在 EJB 的 echo 方法中，访问了资源适配器连接工厂，创建了连接，然后马上关闭了该连接。echo 方法代码如列表 7-3 所示。

列表 7-3 无状态会话 Bean 的 echo 方法（演示访问资源适配器连接工厂）

```
public String echo(String arg)
{
    log.debug("echo, arg="+arg);
    try
    {
        InitialContext iniCtx = new InitialContext();
        Context enc = (Context) iniCtx.lookup("java:comp/env");
        Object ref = enc.lookup("ra/DirContextFactory");
        log.debug("echo, ra/DirContextFactory="+ref);
        DirContextFactory dcf = (DirContextFactory) ref;
        log.debug("echo, found dcf="+dcf);
        DirContext dc = dcf.getConnection();
        log.debug("echo, lookup dc="+dc);
        dc.close();
    }
    catch (NamingException e)
    {
        log.error("Failed during JNDI access", e);
    }
    return arg;
}
```


}

EJB 没有使用 CCI 接口访问资源适配器，而是使用资源适配器专属 API，即 `DirContextFactory` 接口，并将返回 JNDI `DirContext` 作为连接对象。EJB 实例简单地演示了系统契约层，比如查找资源适配器连接工厂、创建资源连接及关闭连接。它没有对该连接做任何处理。既然文件系统是非事务性资源，因此整个实例仅仅演示了资源适配器实现。

在光盘目录运行如下 `ant` 命令能够运行测试程序，其中调用了 `EchoBean.echo`。

```
[nr@toki examples]$ ant -Dchap=chap7 -Dex=1 run-example
Buildfile: build.xml
...
run-example1:
    [copy] Copying 1 file to /tmp/jboss-3.2.3/server/default/deploy
    [echo] Waiting for deploy...
    [java] Created Echo
    [java] Echo.echo('Hello') = Hello

BUILD SUCCESSFUL
Total time: 8 seconds
```

现在来研究资源适配器输出的日志信息，这样开发者能够更好地理解适配器与 JBoss JCA 层的交互。输出信息在 JBoss 服务器发布版的 `server/default/log/server.log` 文件中。接下来，本文将日志文件中看到的事件使用序列图显示出来。

将资源适配器连接工厂提供给应用服务器组件，需要经历若干个步骤。剩下的日志信息描述了客户实例调用 `EchoBean.echo` 方法和该方法与资源适配器连接工厂的交互过程。

图 7-5 为序列图，总结了 `EchoBean` 通过 JNDI 访问资源适配器连接工厂和创建连接过程中出现的事件。

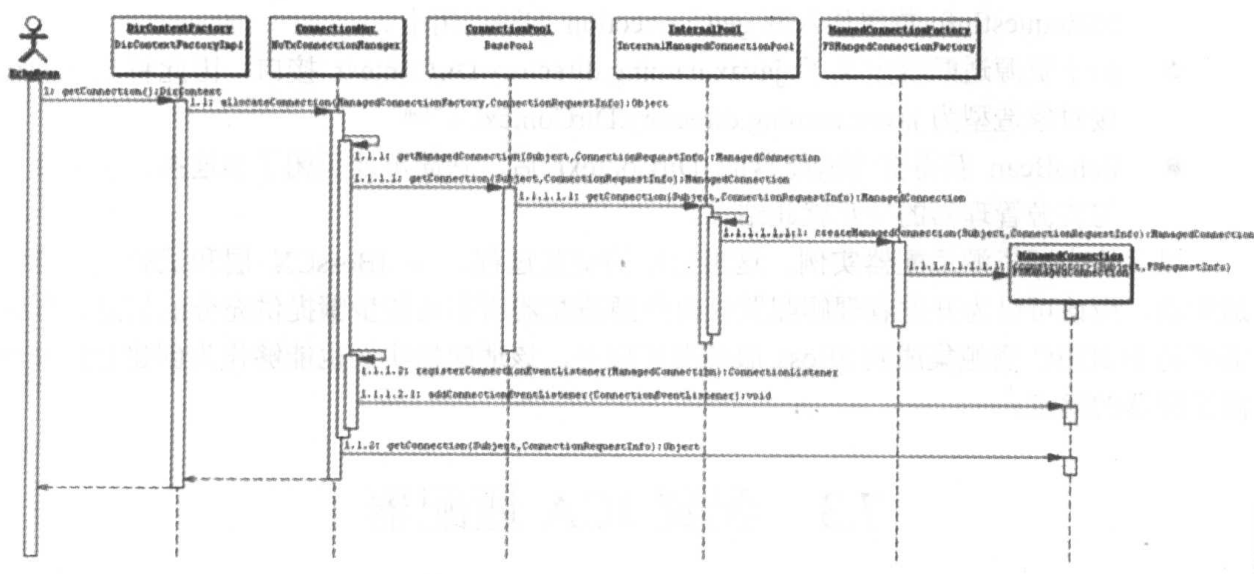


图 7-5 序列图（`EchoBean` 访问资源适配器连接工厂时 JBossCX 框架和适配器的主要交互）

其中，起始点是客户调用 `EchoBean.echo` 方法。为保证序列图的简单性，图中客户直接调用了 `EchoBean.echo` 方法，实际上是 JBoss EJB 容器处理了这个调用。`EchoBean` 与资

源适配器之间存在 3 种不同的交互：查找连接工厂、创建连接及关闭连接。

查找资源适配器连接工厂是由 1.1 序列事件描述的，具体如下：

- 1, echo 方法调用 `getConnection` 方法。其中，`getConnection` 是资源适配器连接工厂的方法，而该资源适配器连接工厂是通过 JNDI 查找获得的，即查找连接到 “java:/NoTransFS” 的 “java:comp/env/ra/DirContextFactory” 名。
- 1.1, `DirContextFactoryImpl` 类要求其关联的 `ConnnectionManager` 分配一连接给它。其中，在 `DirContextFactoryImpl` 的构建阶段，使用了传入的 `ManagedConnectionFactory` 和 `FSRequestInfo`。
- 1.1.1, `ConnectionManager` 用当前 `Subject` 和 `FSRequestInfo` 调用 `getManagedConnection` 方法。
- 1.1.1.1, `ConnectionManager` 请求对象池，申请连接对象。`JBossManagedConnectionPool$BasePool` 获得连接许可后，请求相应的 `InternalPool` 以获得连接。
- 1.1.1.1.1, 既然连接还没有创建，那么该池必须创建新连接。通过请求 `ManagedConnectionFactory` 能够创建新的受管连接。其中，连接池关联的 `Subject` 和 `FSRequestInfo` 数据传入到 `createManagedConnection` 方法调用的参数中。
- 1.1.1.1.1.1, `FSManagedConnectionFactory` 使用传入的 `Subject` 和 `FSRequestInfo` 数据创建新的 `FSManagedConnection` 实例。
- 1.1.1.2, 创建 `javax.resource.spi.ConnectionListener` 实例。`ConnectionManager` 类型决定了创建的监听器类型。这里创建的实例为 `org.jboss.resource.connectionmanager.BaseConnectionManager2@NoTransactionListener` 类型。
- 1.1.1.2.1, 使用 1.2.1.1 创建的 `ManagedConnection` 实例将该监听器注册为 `javax.resource.spi.ConnectionEventListener`。
- 1.1.2, 请求 `ManagedConnection` 连接到底端的资源管理器连接。`Subject` 和 `FSRequestInfo` 数据传入到 `getConnection` 方法调用中。
- 由于资源适配器定义了 `javax.naming.directory.DirContext` 接口，因此将返回的连接对象造型为 `javax.naming.directory.DirContext` 实例。
- `EchoBean` 获得资源适配器的 `DirContext` 后，它简单地关闭了该连接，从而宣告与资源管理器的交互就此结束。

以上给出了资源适配器实例。这里给出的交互过程，即 JBossCX 层和试验性的资源适配器，应该可以为开发者理解配置任何资源适配器所需要的步骤提供充分的信息。如果需要将非 JDBC 资源集成到 JBoss 服务器环境中，该适配器实例也能够作为创建自定义资源适配器的起点。

7.3 配置 JCA 适配器

根据上一节讲述的配置 JBoss JCA 服务及 JCA 资源适配器的实例，本书已经完成 JCA 资源适配器的配置。从 JBoss 3.2 开始，服务器提供了另一种简化模式，它避免了必须指定如此多的配置信息的限制。

7.3.1 配置 JDBC 数据源

在 JBoss 3.2 中, 已经简化了配置 JCA JDBC 连接工厂的语法。JBoss 3.2 不用借助于 MBean 服务描述符配置 MBean 相关的连接管理器工厂, 而是使用简化的数据源描述符。使用 XSL 转换能够将数据源描述符转换成标准的 jboss-service.xml MBean 服务部署描述符, 这里的 XSL 转换由 jboss-jca.sar 部署组件中包含的 org.jboss.deployment.XSLSubDeployer 完成。这种简化的配置描述符同其他可部署组件的部署方式相同, 为了能让 XSLSubDeployer 识别出来, 开发者必须使用 “*-ds.xml” 模式命名描述符。

用于 “*-ds.xml” 配置部署文件顶级 datasources 元素的内容模型, 如图 7-6 所示。

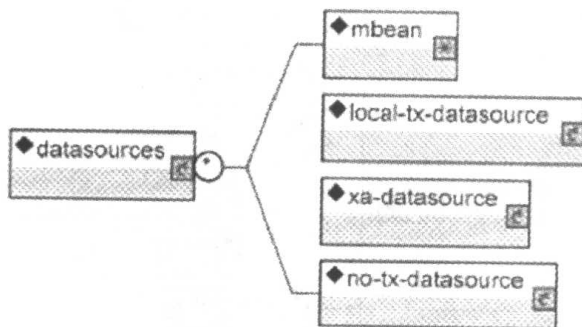


图 7-6 简化的 JCA 数据源配置描述符顶级元素的内容模型

开发者可以在配置部署文件中指定多个数据源配置。datasources 根包含的子元素如下:

- **mbean**: 能够指定若干个 mbean 元素, 以定义 MBean 服务。在完成转换后, 这些 MBean 服务应该包括在 jboss-service.xml 描述符中, datasources 使用它来配置服务。
- **no-tx-datasource**: no-tx-datasource 元素用于指定服务配置, 即 org.jboss.resource.connectionmanager.NoTxConnectionManager。其中, NoTxConnectionManager 是不支持事务的 JCA 连接管理器。no-tx-datasource 子元素的内容模型, 如图 7-7 所示。
- **local-tx-datasource**: local-tx-datasource 元素用于指定服务配置, 即 org.jboss.resource.connectionmanager.LocalTxConnectionManager。LocalTxConnectionManager 实现了 ConnectionEventListener, 而后者又实现 XAResource, 因此能够通过事务管理器来管理事务。为了确保本地事务中的所有工作都是基于同一 ManagedConnection, JBoss 将 xid 添加给 ManagedConnection。当请求 Connection 或事务的连接 handle 在使用中, 它会在全局事务中检查 ManagedConnection 是否已经存在。如果已存在, 则使用它; 如果不存在, 则通过 LocalTransaction 启动空闲的 ManagedConnection, 并使用它。local-tx-datasource 子元素的内容模型, 如图 7-8 所示。
- **xa-datasource**: xa-datasource 元素用于指定服务配置, 即 org.jboss.resource.connectionmanager.XATxConnectionManager。XATxConnectionManager 实现了 ConnectionEventListener, 而后者又实现 XAResource, 因此 ManagedConnection

能够通过事务管理器来管理事务。为了确保本地事务中的所有工作都是基于同一 ManagedConnection, JBoss 将 xid 添加给 ManagedConnection。当请求 Connection 或事务的连接 handle 在使用中, 它会在全局事务中检查 ManagedConnection 是否已经存在。如果已存在, 则使用它; 如果不存在, 则通过 LocalTransaction 启动空闲的 ManagedConnection, 并使用它。xa-datasource 子元素的内容模型, 如图 7-9 所示。

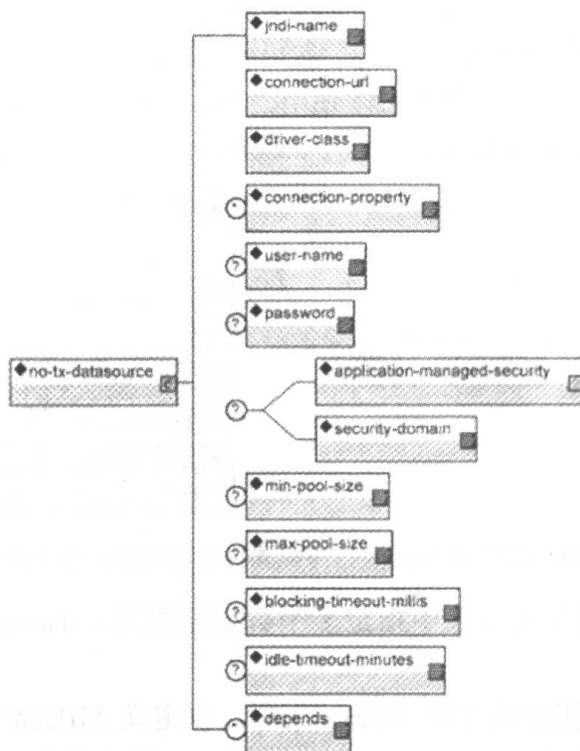


图 7-7 非事务性数据源配置的内容模型

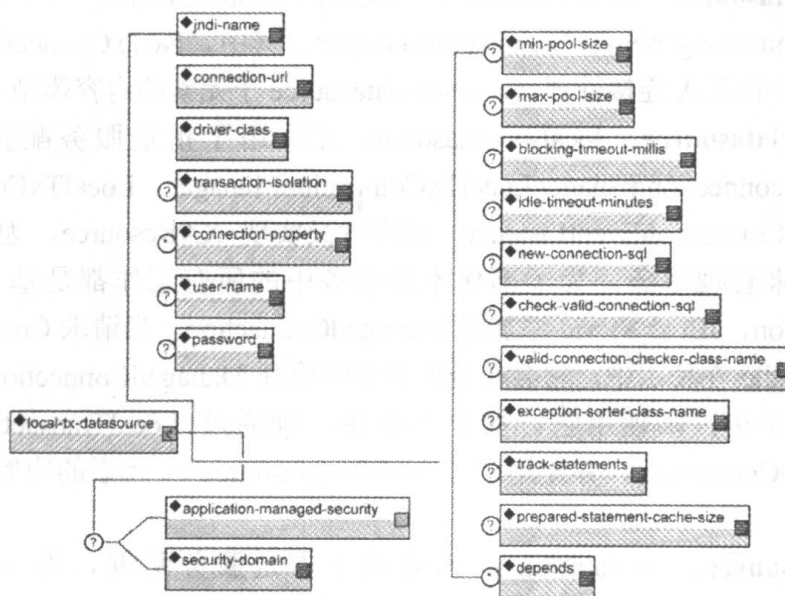


图 7-8 非 XA 数据源配置的内容模型

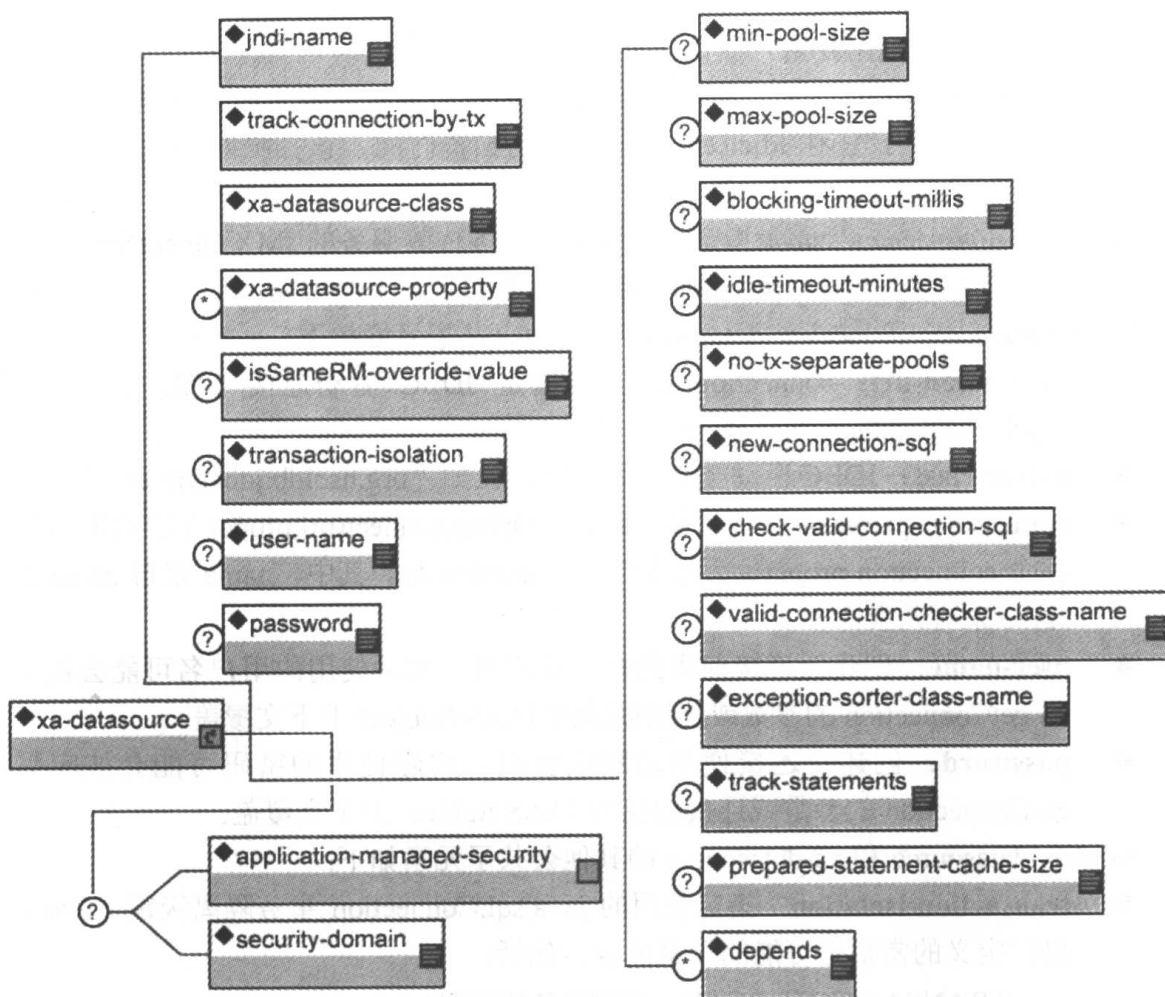


图 7-9 XADataSource 配置的内容模型

所有数据源的公共元素如下：

- **jndi-name**: DataSource 包裹器 (wrapper) 绑定到的 JNDI 名。请注意，该名是相对于“java:/" 前缀的。DataSource 的完整 JNDI 名是：“java:/" + jndi-name。由于 DataSource 在 JBoss 服务器 VM 外部不可用，因此它们被绑定在前缀“java:/" 后。
- **application-managed-security**: 该元素用于指定应用代码提供的参数，比如 getConnection(user,pw)，以区别于池中的连接。
- **security-domain**: 该元素指定是使用应用代码提供的参数，还是基于 JAAS Subject 提供的信息，以区分池中的连接。元素值是用于处理认证的 JAAS 安全性管理器的名字。该名字与 JAAS login-config.xml 描述符中的 application-policy/name 属性关联。
- **min-pool-size**: 表示池中应该持有的最小连接数量。如果第一个连接请求还未到来，是不会创建池实例的。默认值为 0。
- **max-pool-size**: 表示池中允许的最大连接数量。池中存在的连接数不能超过 max-pool-size。默认值为 20。
- **blocking-timeout-millis**: 用于等待连接，但还未抛出异常所能阻塞的最大时间。

请开发者注意，这里的阻塞只是等待连接许可。如果创建新连接消耗了大量的时间，也不会抛出异常。默认值为 5 000。

- **idle-timeout-minutes**: 该属性表明在关闭连接前能空闲的最大时间。实际的最大时间还依赖于空闲 IdleRemover 线程的扫描时间，该扫描时间是所有池中最小 idle-timeout-minutes 的 1/2。
- **depends**: depends 元素指定连接管理器服务依赖服务的 JMX ObjectName 字符串。如果依赖服务没有启动，则连接管理器服务一直不会启动。

no-tx-datasource 和 local-tx-datasource 的其他公共子元素如下：

- **connection-url**: connection-url 元素指定 JDBC 驱动连接 URL 字符串，比如 “jdbc:hsqldb:hsqldb://localhost:1701”。
- **driver-class**: JDBC 驱动类的全限定名，比如 “org.hsqldb.jdbcDriver”。
- **connection-property**: 传递给 java.sql.Driver.connect(url,props)方法的所需属性。各个 connection-property 元素指定 name/value 对。其中，name 来自 name 属性，value 来自元素内容。
- **user-name**: 创建新连接使用的默认用户名。实际使用的用户名可能会被应用代码 getConnection 的参数或连接创建的 JAAS Subject 上下文覆盖。
- **password**: 创建新连接使用的默认密码。实际使用的密码可能会被应用代码 getConnection 的参数或连接创建的 JAAS Subject 上下文覆盖。

local-tx-datasource 和 xa-datasource 的其他公共子元素如下：

- **transaction-isolation**: 指定使用的 java.sql.Connection 事务隔离级别。Connection 接口定义的常量是可能的元素内容，包括：
 - TRANSACTION_READ_UNCOMMITTED
 - TRANSACTION_READ_COMMITTED
 - TRANSACTION_REPEATABLE_READ
 - TRANSACTION_SERIALIZABLE
 - TRANSACTION_NONE
- **new-connection-sql**: 创建新连接时，期待执行的 SQL 语句。它能够用于配置具体数据库的连接信息，而这种连接信息是不能通过连接属性配置的。
- **check-valid-connection-sql**: 在从连接池获得返回连接前应该运行的 SQL 语句，以测试连接的有效性，这对于检查陈旧池连接很有帮助。比如，示例语句 “select count(*) from x”。
- **exception-sorter-class-name**: org.jboss.resource.adapter.jdbc.ExceptionSorter 接口的实现类名，用于过滤 SQLException，从而能够判断是否有连接错误事件产生。目前的实现有：
 - org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter
 - org.jboss.resource.adapter.jdbc.vendor.SybaseExceptionSorter
- **valid-connection-checker-class-name**: valid-connection-checker-class-name 元素指定 org.jboss.resource.adapter.jdbc.ValidConnectionChecker 接口的实现类名。该类将调用 SQLException isValidConnection(Connection e)方法，从而判断从连接池返回的

连接是否有效。它将覆盖 `check-valid-connection-sql` 值。目前的实现有“`org.jboss.resource.adapter.jdbc.vendor.OracleValidConnectionChecker`”。

- **track-statements**: boolean 标志位, 用于检查连接归还给池时是否还有打开的 statement。如果为 true, 则每出现一个未关闭的 statement, 就会抛出一条警告信息。如果 Log4j category `org.jboss.resource.adapter.jdbc.WrappedConnection` 生效到 trace 级, 则连接关闭调用日志信息也将记录下来。在实际产品运行环境中, 需要关闭该特征。
- **prepared-statement-cache-size**: 用于 SQL 查询的 LRU 缓存中每个连接的预编译语句的数量。如果设为 0, 则不提供缓存。

单个 `xa-datasource` 的其他子元素如下:

- **track-connection-by-tx**: 如果指定为 true; 则连接管理器将为连接保存一个 `xid`, 当事务结束时才将连接归还给连接池, 并关闭或分离 (方法调用返回) 所有的连接 handle。其负面影响是, 并不会挂起和恢复连接的 `XAResource` 的 `xid`。本地事务也是使用同样的连接跟踪方式。

XA 规范指出, 在任何线程 (如有需要, 挂起其他事务) 运行的任何时间都可能使用 `xid` 登记任何事务的连接。初始的 JCA 实现确实也是这样的, 即只要连接控制权交给了 EJB 或连接 handle 已关闭, 则会强制清除连接, 并将它们归还给连接池。其他事务下次可以使用该连接, 因此使用该连接的事务必须结束, 从而无法取回原始连接。很明显, 大部分 `XADataSource` 驱动厂商并不支持该特性, 它们要求所有的工作都必须在具有特定 `xid` 的相同连接下完成。

- **xa-datasource-class**: `xa-datasource-class` 元素指定, `javax.sql.XADataSource` 实现的全限定类名。比如“`com.informix.jdbcx.IfxXADataSource`”。
- **xa-datasource-property**: `xa-datasource-property` 元素指定, `XADataSource` 实现类的属性。通过 `name` 属性给出属性名, 属性值通过 `xa-datasource-property` 元素内容给出。这些属性都是通过寻找 JavaBean 风格 `getter` 方法找到的。如果找到, 则使用 `java.beans.PropertyEditor` 将字符串转换为属性的真正类型。
- **isSameRM-override-value**: boolean 标志位, 它能够覆盖 XA 受管连接的 `javax.transaction.xa.XAResource.isSameRM(XAResource xaRes)` 方法的行为。如果指定该元素, 则它将一直作为 `isSameRM(xaRes)` 的返回值, 而不会理睬 `xaRes` 参数的取值。
- **no-tx-separate-pools**: `no-tx-separate-pools` 元素表明需要两个连接池。一个用于有 JTA 事务的连接工作, 另一个用于没有 JTA 事务的连接工作。如果一直没有客户请求, 则不会创建这些池。比如 Oracle (也可能存在其他厂商) XA 实现, 它允许通过或者不通过 JTA 事务来使用 XA 连接。

7.3.2 配置常见 JCA 适配器

XSLSubDeployer 也支持使用简化的语法来部署非 JDBC JCA 资源适配器。“`*-ds.xml`”配置部署文件的顶级连接工厂元素的内容模型如图 7-10 所示。

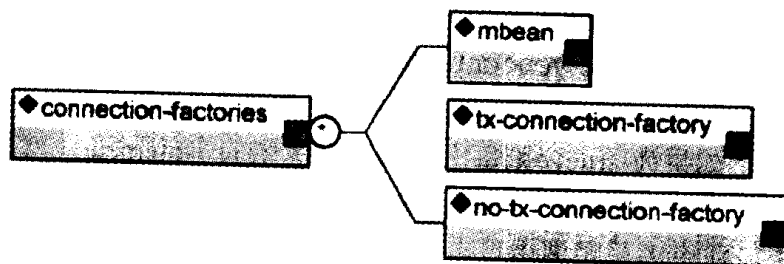


图 7-10 简化的 JCA 适配器连接工厂配置描述符顶级元素的内容模型

可以在该配置部署文件中指定多个连接工厂配置，connection-factories 根的子元素如下：

- **mbean**：可以指定若干个 mbean 元素，以定义 MBean 服务。其中，经过转换后，这些 MBean 服务将包括在 jboss-service.xml 描述符中，适配器可以使用它来配置服务。
- **no-tx-connection-factory**：no-tx-connection-factory 元素用于指定服务配置，即 org.jboss.resource.connectionmanager.NoTxConnectionManager，NoTxConnectionManager 是不支持事务能力的 JCA 连接管理器。no-tx-connection-factory 的子元素内容模型，如图 7-11 所示。
- **tx-connection-factory**：tx-connection-factory 元素用于指定服务配置，即 org.jboss.resource.connectionmanager.TxConnectionManager。tx-connection-factory 的子元素内容模型，如图 7-12 所示。

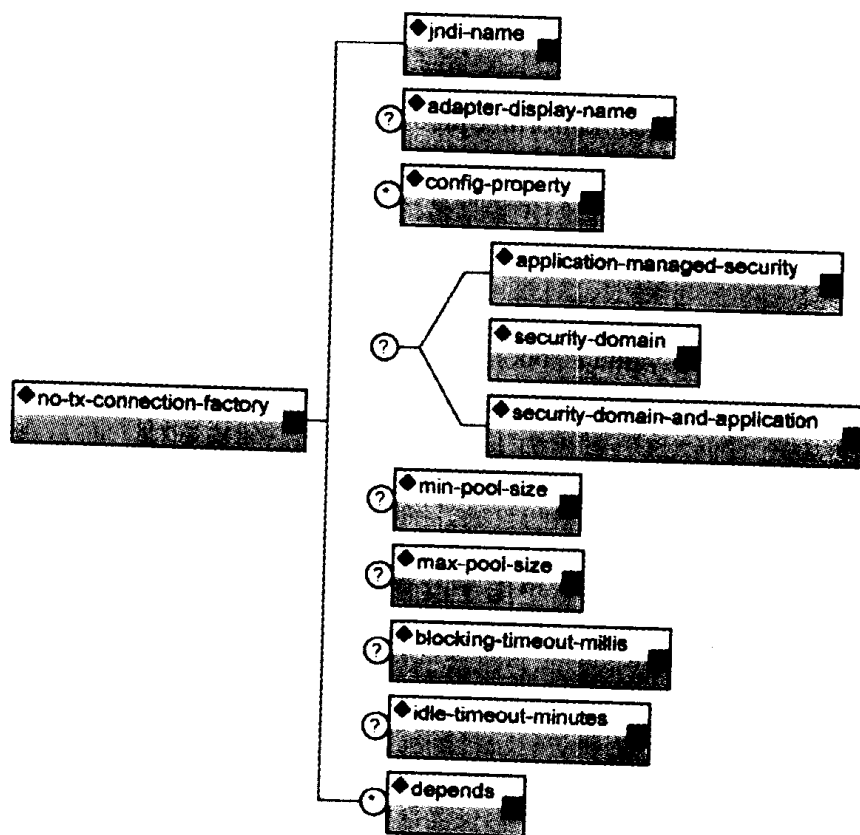


图 7-11 no-tx-connection-factory 元素的内容模型

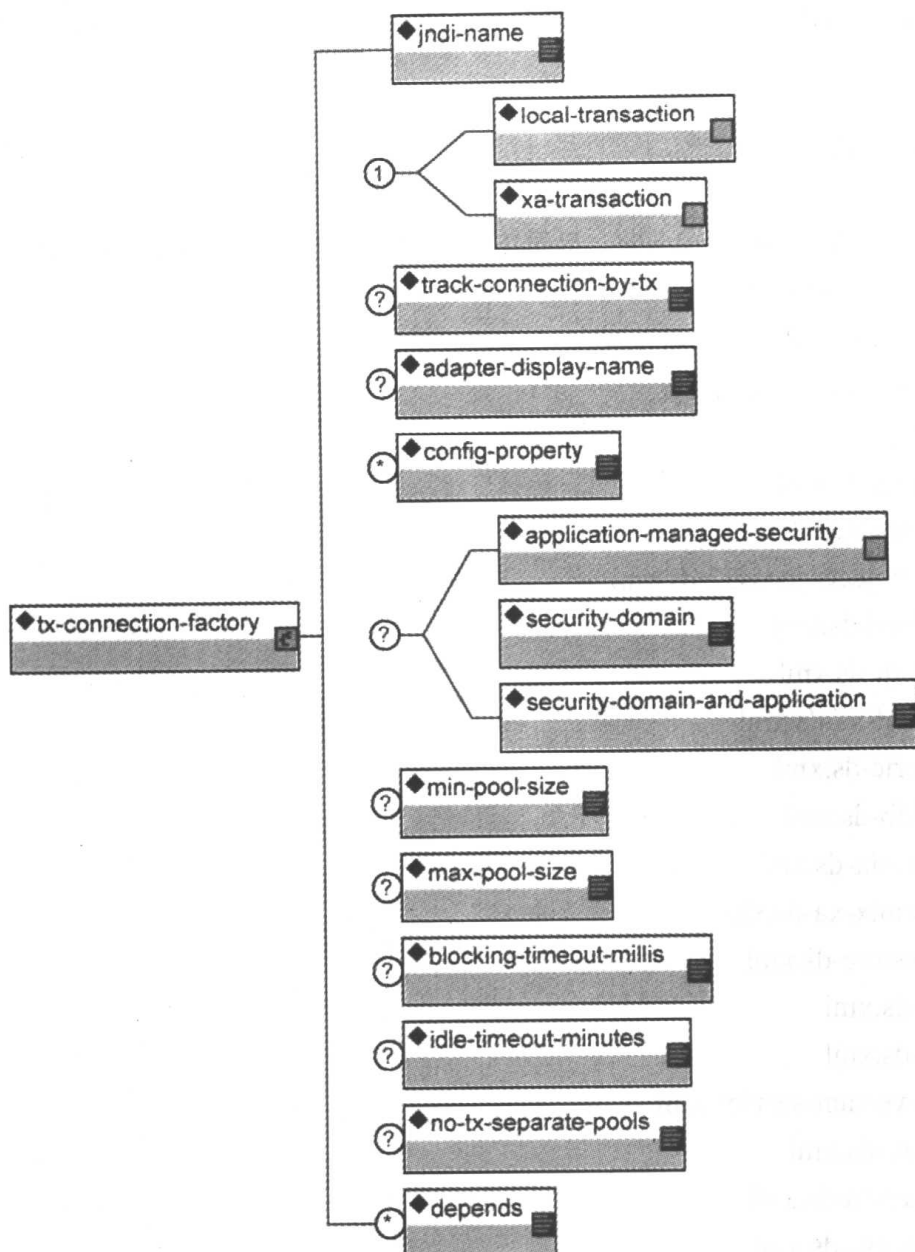


图 7-12 tx-connection-factory 元素的内容模型

大部分元素同数据源配置相同，不同的元素如下：

- **adaptor-display-name:** adaptor-display-name 元素指定，便于开发者阅读的、用于连接管理器 MBean 服务的名字。
- **config-property:** config-property 元素指定，提供给 ManagedConnectionFactory (MCF) MBean 服务配置的若干属性。各个 config-property 元素指定 MCF 属性值。其中，config-property 元素要求提供如下两个属性：
 - name: 属性名
 - type: 属性的全限定类型

config-property 元素内容给出了属性值的字符串表示。借助于 PropertyEditor 能够将它转换为属性的真正类型。

- **local-transaction** | **xa-transaction**: 这些元素指定 tx-connection-factory 支持本地事务还是 XA 事务。

7.3.3 配置实例

JBOSS_DIST/docs/examples/jca 目录下包含了许多第三方 JDBC 驱动的配置实例，具体包括如下一些配置实例。

- asapxcess-jb3.2-ds.xml
- cicsr9s-service.xml
- db2-ds.xml
- db2-xa-ds.xml
- facets-ds.xml
- fast-objects-jboss32-ds.xml
- firebird-ds.xml
- firstsql-ds.xml
- firstsql-xa-ds.xml
- generic-ds.xml
- hsqldb-ds.xml
- informix-ds.xml
- informix-xa-ds.xml
- jdatastore-ds.xml
- jms-ds.xml
- jsql-ds.xml
- lido-versant-service.xml
- mimer-ds.xml
- mimer-xa-ds.xml
- msaccess-ds.xml
- mssql-ds.xml
- mssql-xa-ds.xml
- mysql-ds.xml
- oracle-ds.xml
- oracle-xa-ds.xml
- postgres-ds.xml
- sapdb-ds.xml
- sapr3-ds.xml
- solid-ds.xml
- sybase-ds.xml

第 8 章 JBoss 之安全性

——J2EE 安全性配置和架构

安全性是任何企业应用的基本组成部分。企业应用需要约束访问应用用户和控制应用用户所能完成的操作类型。J2EE 规范为 EJB 和 Web 组件定义了简单的、基于角色的安全性模型。JBoss 中处理安全性的组件框架称之为 JBossSX 扩展框架。JBossSX 安全性扩展既能支持基于角色的 J2EE 安全性声明模型，又能支持借助于安全性代理层的自定义安全集成。安全性声明模型的默认实现是基于 Java 认证和授权服务（Java Authentication and Authorization Service, JAAS）登录模块和 Subject 的。安全性代理层允许将自定义安全性以独立于 EJB 业务对象的方式添加给 EJB，这里的自定义安全性指的是那些不能够使用声明模型（declarative model）的安全性。在深入讨论 JBoss 安全性实现细节前，本章首先为开发者介绍相关的基础概念，即 EJB 2.0、Servlet 2.2 规范安全性模型及 JAAS。

8.1 J2EE 安全性声明概述

J2EE 规范提倡的安全性模型是声明模型。这种声明体现在，组件使用标准的 XML 描述符描述安全性角色和许可，而不是将安全性嵌入在业务组件中。因此，它将安全性从业务级代码中隔离开，因为安全性只是为部署组件添加的新功能，而不是业务逻辑层面的内容。例如，考虑某用于访问银行账号的 ATM 组件，其安全性需求、角色以及许可基本上跟用户如何访问银行账号、哪间银行管理账号、ATM 机的具体位置等无关。

开发者借助于标准的 J2EE 部署描述符能够为 J2EE 应用提供基于应用安全性需求规范的保护。开发者通过使用 `ejb-jar.xml` 和 `web.xml` 部署描述符能够保护企业应用中对 EJB 和 Web 组件的访问。图 8-1 和图 8-2 分别给出了 EJB 2.0 和 Servlet 2.2 部署描述符中的安全性相关元素。

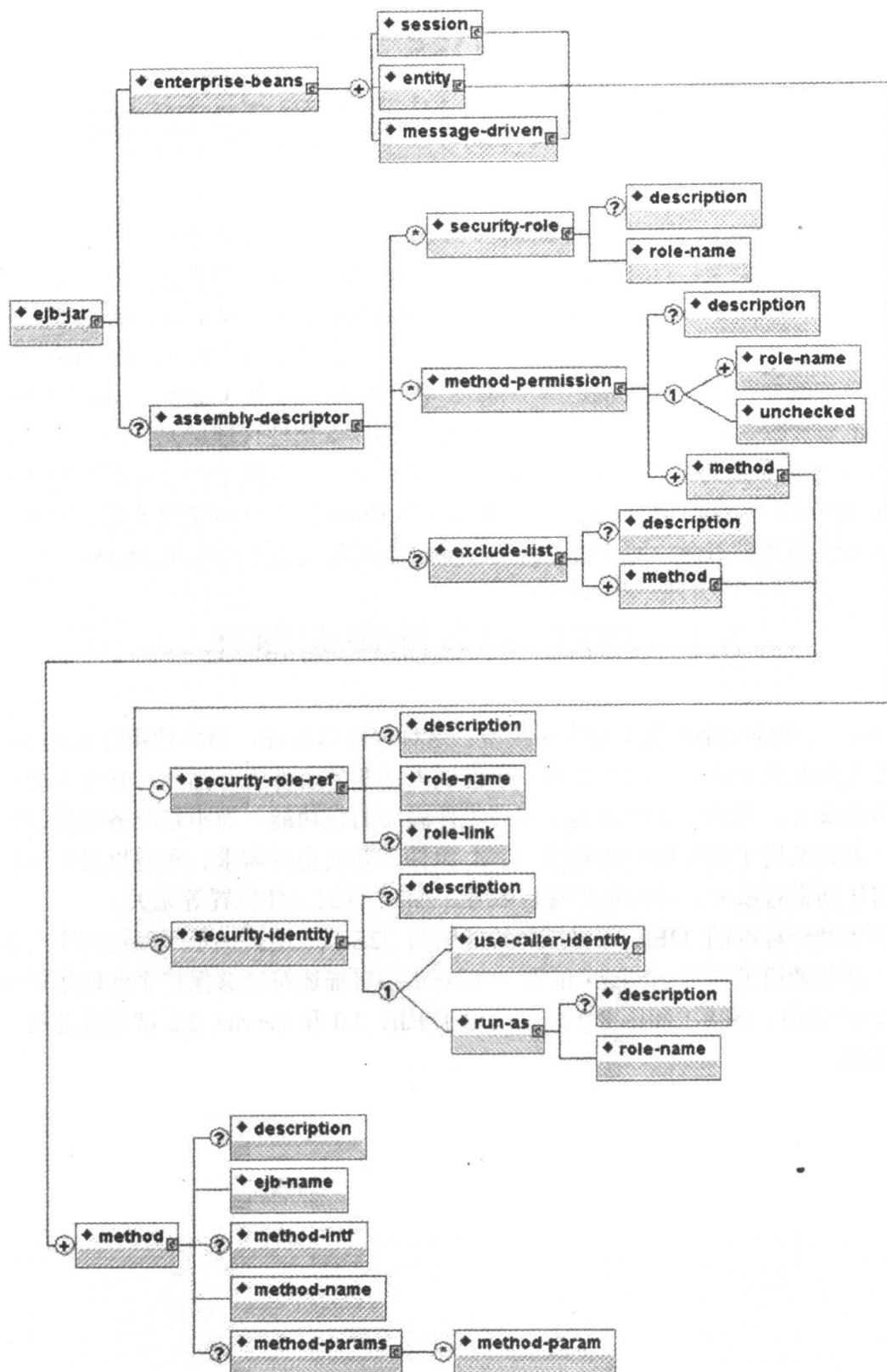


图 8-1 EJB 2.0 部署描述符的内容模型（仅显示安全性相关元素）

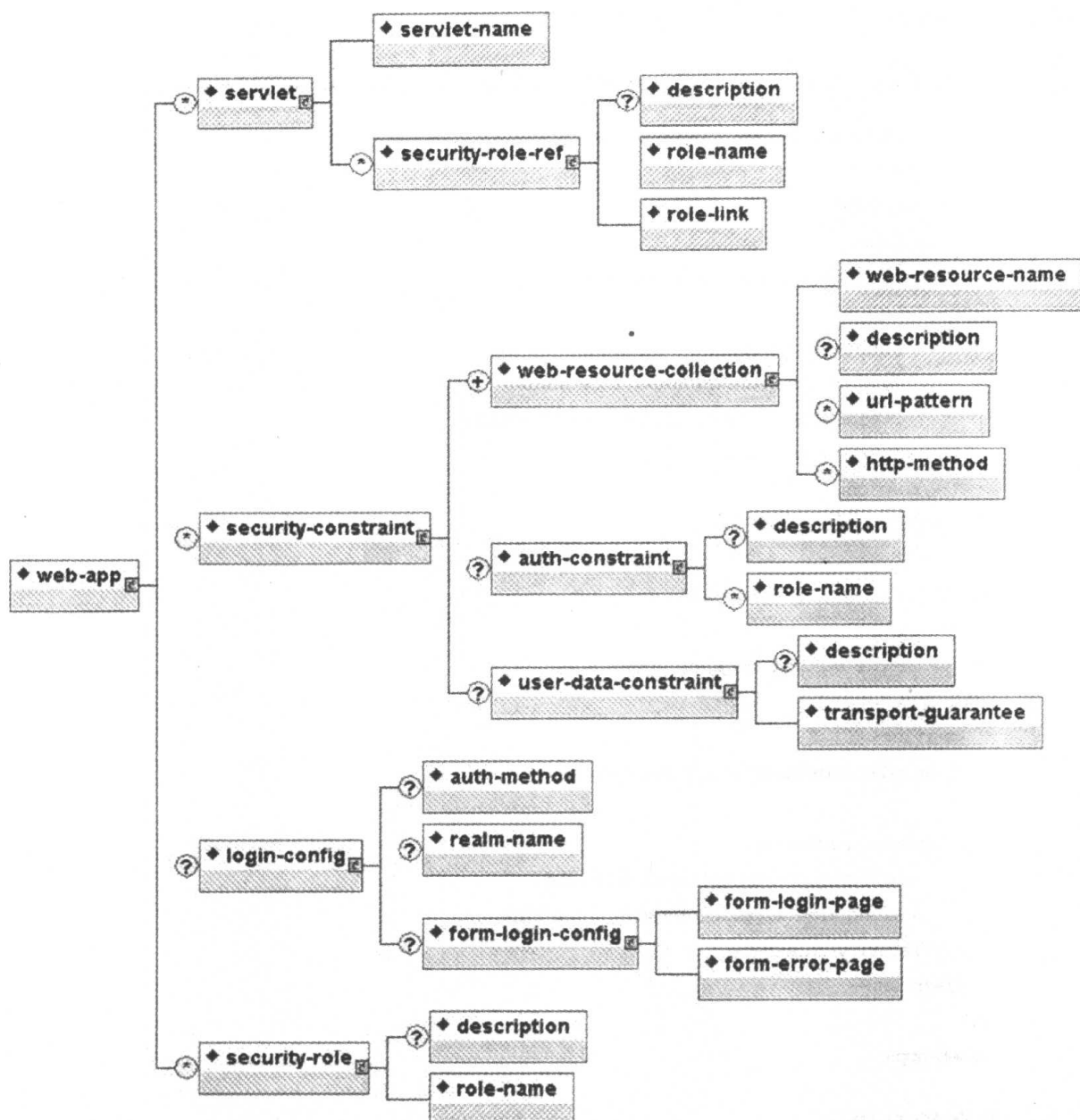


图 8-2 Servlet 2.2 部署描述符的内容模型（仅显示安全性相关元素）

接下来，本节内容阐述图 8-1 和图 8-2 给出的各种安全性元素的具体目的和使用。

8.1.1 安全性引用

EJB 和 Servlet 都能够声明一个或多个 security-role-ref 元素。该元素将触发组件使用 role-name 值作为 isCallInRole(String)方法的参数。通过使用 isCallInRole 方法，组件能够验证调用者的角色是否已在 security-role-ref/role-name 元素中给出声明。role-name 元素值通过 role-link 元素连接到 security-role 元素。当不能使用 method-permissions 元素定义的角色进行安全性验证时，开发者一般都需要借助于 isCallerInRole 方法。然而，本书不推荐开发者使用 isCallInRole 方法，因为它将安全性逻辑嵌入在组件代码中。列表 8-1 给出了

包含 security-role-ref 元素的描述符实例片段。

列表 8-1 包含 security-role-ref 元素用法的 ejb-jar.xml 和 web.xml 描述符片段

```
<!-- A sample ejb-jar.xml fragment -->
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>ASessionBean</ejb-name>
      ...
      <security-role-ref>
        <role-name>TheRoleICheck</role-name>
        <role-link>TheApplicationRole</role-link>
      </security-role-ref>
    </session>
  </enterprise-beans>
  ...
</ejb-jar>

<!-- A sample web.xml fragment -->
<web-app>
  <servlet>
    <servlet-name>AServlet</servlet-name>
    ...
    <security-role-ref>
      <role-name>TheServletRole</role-name>
      <role-link>TheApplicationRole</role-link>
    </security-role-ref>
  </servlet>
  ...
</web-app>
```

8.1.2 安全性身份

EJB 可以声明可选 security-identity 元素。它是 EJB 2.0 规范才引入的元素，其主要功能是指定当调用其他组件的方法时，EJB 应该使用何种身份。调用身份可能是当前调用者的本来身份，或者特定角色。应用集成者使用带有 user-caller-identity 子元素的 security-identity 元素将当前调用者的身份传播给 EJB 中的方法调用。如果没有显式地给出 security-identity 元素声明，默认情况下也会传递调用者的身份。

另外，应用集成者能够使用 run-as/role-name 子元素指定如下内容，即 role-name 值给出的特定安全性角色应该用做 EJB 中方法调用的安全性身份。请注意，这并没有改变调用者的身份，开发者通过 EJBContext.getCallerPrincipal() 可以验证。但是，调用者的安全性角色却变成了 run-as/role-name 元素值指定的单个角色。比如，某使用 run-as 元素的情形如下，即为阻止外部客户访问内部 EJB 而提供保护功能。与此同时，开发者还可以为内部 EJB 使用 method-permission 元素，以限制访问角色不会分配给外部客户，于是配置

了 run-as/role-name 元素的、需要使用内部 EJB 的 EJB 同受限角色相同。列表 8-2 给出了包含 security-identity 元素的描述符片段。

列表 8-2 包含 security-identity 元素的 ejb-jar.xml 描述符实例片段

```
<!-- A sample ejb-jar.xml fragment -->
<ejb-jar>
<enterprise-beans>
  <session>
    <ejb-name>ASessionBean</ejb-name>
    ...
    <security-identity>
      <use-caller-identity/>
    </security-identity>
  </session>
  <session>
    <ejb-name>RunAsBean</ejb-name>
    ...
    <security-identity>
      <run-as>
        <description>A private internal role</description>
        <role-name>InternalRole</role-name>
      </run-as>
    </security-identity>
  </session>
</enterprise-beans>
...
</ejb-jar>
```

另外，J2EE 中的 Servlet 2.3 规范也引入了安全性身份，但 JBoss 3.0 目前还未对它提供支持。

8.1.3 安全性角色

使用 security-role-ref 或 security-identity 元素引用的安全性角色名需要映射到应用的角色声明中。应用集成者使用 security-role 元素定义逻辑安全性角色。role-name 值也是逻辑应用角色名，比如 Administrator、Architect、SalesManager 等。



何为角色？J2EE 规范强调，部署描述符中的安全性角色是用于定义应用的逻辑安全性视图的。开发者不要混淆定义在 J2EE 部署描述符中的角色和组（group）、用户、Principal 及目标企业操作环境中的其他概念。部署描述符角色是构建应用中的具体应用域名。比如，银行应用可以使用角色名 BankManager、Teller 及 Customer。

JBoss 中 security-role 元素只用于映射 security-role-ref/role-name 值到组件角色引用的逻辑角色。应用的安全性管理器能够动态改变分配给用户的角色，本章在介绍 JBossSX 实

现细节时将有其详细论述。为了能够声明方法许可，部署在 JBoss 中的应用可以不定义 security-role 元素。因此，考虑到应用服务器移植性和部署描述符的可维护性，开发者给出 security-role 元素只是一种良好的操作风格。列表 8-3 给出了包含 security-role 元素的描述符实例片段。

列表 8-3 包含 security-role 元素的 ejb-jar.xml 和 web.xml 描述符实例片段

```
<!-- A sample ejb-jar.xml fragment -->
<ejb-jar>
...
<assembly-descriptor>
  <security-role>
    <description>The single application role</description>
    <role-name>TheApplicationRole</role-name>
  </security-role>
</assembly-descriptor>
</ejb-jar>

<!-- A sample web.xml fragment -->
<web-app>
...
<security-role>
  <description>The single application role</description>
  <role-name>TheApplicationRole</role-name>
</security-role>
</web-app>
```

8.1.4 EJB 方法许可

通过声明 method-permission 元素，应用集成者能够设置角色，以限制对 EJB home 和远程接口的访问。各个 method-permission 元素含有一个或多个 role-name 子元素，这些子元素用于定义访问 EJB 方法的逻辑角色。其中，开发者通过 method 子元素能够定义上述受限的 EJB 方法。从 EJB 2.0 开始，应用集成者可以声明 unchecked 元素（而不是 role-name 元素），使得任何通过认证的用户都能够访问 method 子元素指定的方法。另外，应用集成者也可以使用 exclude-list 元素声明任何客户都不能够访问的方法。对于那些 method-permission 元素声明中没有给出的 EJB 方法而言，其默认情况下是不允许客户使用的，视同于将这些方法放置到 exclude-list 元素。

EJB 规范允许 EJB 应用中存在 3 种声明 method 元素的风格。

- 风格 1，用于引用指定企业 Bean 的 home 和组件接口中的所有方法。

```
<method>
  <ejb-name>EJBNAME</ejb-name>
  <method-name>*</method-name>
</method>
```

- 风格 2，用于引用指定企业 Bean 中 home 或组件接口的特定方法。如果接口中存在多个重复的指定方法，风格 2 也引用这些重复方法。

```
<method>
  <ejb-name>EJBNAME</ejb-name>
  <method-name>METHOD</method-name>
</method>
```

- 风格 3，用于从多个重复方法集合中引用特定方法。该方法必须定义在企业 Bean 的 home 或远程接口中，method-param 元素值指定对应方法参数类型的全限定名。如果存在多个重复的指定方法，风格 3 也引用这些方法。

```
<method>
  <ejb-name>EJBNAME</ejb-name>
  <method-name>METHOD</method-name>
  <method-params>
    <method-param>PARAMETER_1</method-param>
    ..
    <method-param>PARAMETER_N</method-param>
  </method-params>
</method>
```

可选 method-intf 元素能够区分企业 Bean 中 Home 和远程接口定义中具有相同名字和参数类型的方法。列表 8-4 给出了包含 method-permission 元素的 ejb-jar.xml 描述符实例片段。

列表 8-4 包含 method-permission 元素的 ejb-jar.xml 描述符实例片段

```
<ejb-jar>
  <assembly-descriptor>
    <method-permission>
      <description>The employee and temp-employee roles may
        access any method of the EmployeeService bean
      </description>
      <role-name>employee</role-name>
      <role-name>temp-employee</role-name>
      <method>
        <ejb-name>EmployeeService</ejb-name>
        <method-name>*</method-name>
      </method>
    </method-permission>

    <method-permission>
      <description>The employee role may access the
        findByPrimaryKey, getEmployeeInfo, and the
        updateEmployeeInfo(String) method of the AardvarkPayroll
```

```

        bean
    </description>
    <role-name>employee</role-name>
    <method>
        <ejb-name>AardvarkPayroll</ejb-name>
        <method-name>findByPrimaryKey</method-name>
    </method>

    <method>
        <ejb-name>AardvarkPayroll</ejb-name>
        <method-name>getEmployeeInfo</method-name>
    </method>

    <method>
        <ejb-name>AardvarkPayroll</ejb-name>
        <method-name>updateEmployeeInfo</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
        </method-params>
    </method>
</method-permission>

<method-permission>
    <description>The admin role may access any method
        of the EmployeeServiceAdmin bean
    </description>
    <role-name>admin</role-name>
    <method>
        <ejb-name>EmployeeServiceAdmin</ejb-name>
        <method-name>*</method-name>
    </method>
</method-permission>

<method-permission>
    <description>Any authenticated user may access any method
        of the EmployeeServiceHelp bean
    </description>
    <unchecked/>
    <method>
        <ejb-name>EmployeeServiceHelp</ejb-name>
        <method-name>*</method-name>
    </method>
</method-permission>
<exclude-list>
    <description>No fireTheCTO methods of the EmployeeFiring
        bean may be used in this deployment
    </description>

```

```

</description>
<method>
  <ejb-name>EmployeeFiring</ejb-name>
  <method-name>fireTheCTO</method-name>
</method>
</exclude-list>
</assembly-descriptor>
</ejb-jar>

```

8.1.5 Web 内容安全性约束

在 Web 应用中，通过角色定义的安全性能够保护 URL 模式的内容访问，这些 URL 模式标识了受保护内容，应用集成者使用 web.xml 中的 security-constraint 元素声明能够达到上述目的。开发者使用一个或多个 web-resource-collection 元素能够保护访问内容，各个 web-resource-collection 元素包含可选的若干个 url-pattern 和 http-method 元素，url-pattern 位于 http-method 元素前。url-pattern 元素值指定的 URL 模式表明，请求 URL 必须匹配试图访问的保护内容。http-method 元素值指定 HTTP 请求类型。

可选 user-data-constraint 元素指定客户到服务器连接的传输层要求。该要求是用于内容一致性（防止数据传输过程中被篡改），或用于机密性（防止传输过程中发生读操作）。transport-guarantee 元素值指定客户端和服务器连接之间通信的保护程度，具体取值范围如下：NONE、INTEGRAL 或 CONFIDENTIAL。其中，NONE 值意味着应用不需要任何传输保护；INTEGRAL 值意味着保证数据在传输过程中不会发生改动；CONFIDENTIAL 值意味着数据在传输过程中不能被其他实体阅读。在通常情况下，使用 INTEGRAL 或 CONFIDENTIAL 标志表明传输层需要使用 SSL。

可选 login-config 元素用于配置应该使用的认证方法、应用应该使用的域（realm）名及表单登录方式中所需的属性。auth-method 子元素指定 Web 应用的认证方式。在允许用户访问被授权约束保护的 Web 资源之前，Web 服务器首要完成的工作是使用已配置机制认证用户。auth-method 子元素的合法取值如下：BASIC、DIGEST、FORM 或 CLIENT-CERT。其中，realm-name 子元素指定 HTTP BASIC 和 DIGEST 授权过程中使用的 realm 名，form-login-config 子元素指定基于表单登录中应该使用的登录和错误页面。如果 auth-method 子元素值不是 FORM，则 Web 服务器将忽略 form-login-config 及其子元素。

列表 8-5 给出了包含 security-constraint 和相关元素的 web.xml 描述符实例片段。其中，访问 Web 应用“/restricted”路径下任何 URL 的角色必须是 AuthorizedUser。它没有传输保护要求，用于获得用户身份的认证方法是 BASIC HTTP 认证。

列表 8-5 给出了包含 security-constraint 和相关元素的 web.xml 描述符实例片段

```

<web-app>
...
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Secure Content</web-resource-name>

```

```
<url-pattern>/restricted/*</url-pattern></>
<web-resource-collection>
  <auth-constraint>
    <role-name>AuthorizedUser</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>NONE</transport-guarantee>
  </user-data-constraint>
</security-constraint>
...
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>The Restricted Zone</realm-name>
</login-config>
...
<security-role>
  <description>The role required to access restricted content
</description>
  <role-name>AuthorizedUser</role-name>
</security-role>
</web-app>
```

8.1.6 使用 JBoss 中的安全性声明

上述给出的 J2EE 安全性元素只是从应用角度提出的安全性要求。既然 J2EE 安全性元素只是声明逻辑角色，则应用部署者需要将应用域中的角色映射到部署环境中。J2EE 规范忽略了具体应用服务器的这些具体细节。在 JBoss 中，如果需要映射应用角色到部署环境，必须指定安全性管理器。其中，安全性管理器借助于 JBoss 服务器特定部署描述符实现了 J2EE 安全性模型，本节暂时不讨论它。在“8.3 JBoss 安全性模型”节中，在讨论普通 JBoss 服务器安全性接口时，将有安全性配置的详细介绍。

8.2 JAAS 介绍

默认 JBossSX 框架是基于 JAAS API 实现的。由于 JAAS API 是较新的 API，目前还未广泛使用，因此要掌握 JBossSX 的实现细节，开发者需要熟悉 JAAS API 的基本元素。本节主要介绍 JAAS，从而为讨论 JBossSX 架构作准备。

JAAS 包的其他细节，可参考 JAAS 主页：<http://java.sun.com/products/jaas/>。

何为 JAAS

用于用户认证和授权的 JAAS 1.0 API 是由一套 Java 包组成的，实现了标准的插入式认证模块（Pluggable Authentication Module, PAM）框架，并适当地扩展了 Java 2 平台的

访问控制架构，以支持基于用户的授权。JAAS 初次是以 JDK 1.3 的扩展包形式发布的，从 JDK 1.4 开始就一直绑定在标准 JDK 中。由于 JBossSX 框架只使用了 JAAS 的认证功能来实现基于角色的 J2EE 安全性声明模型，因此本节暂且只关注认证功能。

本节的大部分材料都是来自于 JAAS 1.0 开发者指南。因此，如果开发者已经熟悉这部分内容，则可以直接跳到“8.4 JBoss 安全性扩展架构”节，以开始研究 JBossSX 架构。

JAAS 认证是一种插件式实现。基于此，Java 应用和底层的认证技术相互独立，并允许 JBossSX 安全性管理器能够工作在不同的安全性基础框架服务中。开发者不需要变更 JBossSX 安全性管理器实现，就能够实现安全性基础框架服务的集成；开发者需要变更的只是配置 JAAS 使用的认证栈。

JAAS 核心类

开发者可以将 JAAS 核心类分成 3 种类型：公共、认证及授权。下面只列举出了公共类和认证类，因为 JBossSX 功能实现仅仅使用了这两种类型。

- 公共类：
 - Subject (javax.security.auth.Subject)
 - Principal (java.security.Principal)
- 认证类：
 - Callback (javax.security.auth.callback.Callback)
 - CallbackHandler (javax.security.auth.callback.CallbackHandler)
 - Configuration (javax.security.auth.login.Configuration)
 - LoginContext (javax.security.auth.login.LoginContext)
 - LoginModule (javax.security.auth.spi.LoginModule)

(1) Subject 和 Principal

为授权对资源的访问，应用系统首先需要认证请求源。JAAS 框架将术语 Subject 定义为请求源。Subject 类是 JAAS 的核心类，它表示了单个实体的信息，比如某人或服务。它包括实体的 Principal、公共凭证及私有凭证。JAAS API 使用 Java 2 现有 java.security.Principal 接口表示 Principal，Principal 仅仅是类型化名。

在认证过程中，应用系统必须为 Subject 提供相关身份或 Principal。Subject 可能含有多个 Principal。比如，某人名为 John Doe (Principal)，社会保险号为 123-45-6789 (Principal)，用户名为 johnd (Principal)，所有这些信息都能帮助区别不同的 Subject。开发者可以通过如下两种方法获取 Subject 的 Principal。

```
{  
...  
    public Set getPrincipals() {...}  
    public Set getPrincipals(Class c) {...}  
}
```

前一个方法返回 Subject 含有的所有 Principal，后一个方法只返回 Class c 或其子类实例的 Principal。如果 Subject 没有匹配的 Principal，则返回一个空集合。需要注意的是，java.security.acl.Group 是 java.security.Principal 的子接口，因此 Principal 集合中的实例有可

能表示 Principal 组或其他 Principal 的逻辑分组。

（2）Subject 认证

Subject 认证要求 JAAS 登录，具体过程如下：

步骤

（1）应用使用登录配置名实例化 LoginContext 对象，并使用 CallbackHandler 实例操作 Callback 对象，这也是 LoginModule 配置要求完成的内容。

（2）LoginContext 告知 Configuration 去装载指定登录配置中所包括的所有 LoginModule。如果指定登陆配置不存在，则使用默认配置，即“other”配置。

（3）应用调用 LoginContext.login 方法。

（4）login 方法调用所有已装载的 LoginModule。当每个 LoginModule 试图认证 Subject 时，它会调用关联 CallbackHandler 的 handle 方法，从而获得认证过程所要求的信息。接下来，要求的信息会以 Callback 对象数组的形式传给 handle 方法。一旦认证成功，LoginModule 将为 Subject 提供相关的 Principal 和凭证。

（5）LoginContext 将认证状态返回给应用。如果从 login 方法返回，表示认证成功。如果从 login 方法抛出 LoginException，表示认证失败。

（6）如果认证成功，应用将调用 LoginContext.getSubject 方法获得已认证的 Subject。

（7）Subject 认证结束后，应用通过调用 LoginContext.logout 方法能够删除如下内容，即 login 方法为 Subject 提供的所有 Principal 和相关认证资料。

LoginContext 类为认证 Subject 提供了基本的方法，同时也提供了一种独立于底层认证技术的应用开发方式。LoginContext 委派 Configuration 判断具体应用的已配置认证服务。其中，LoginModule 类表示认证服务。因此，开发者不用修改应用本身，就能够将不同 LoginModule 插入到应用中。列表 8-6 给出的代码片段可以演示应用完成 Subject 的认证所需要的步骤。

列表 8-6 认证过程步骤（从应用角度考虑）

```
CallbackHandler handler = new MyHandler();
LoginContext lc = new LoginContext("some-config", handler);
try
{
    lc.login();
    Subject subject = lc.getSubject();
}
catch(LoginException e)
{
    System.out.println("authentication failed");
    e.printStackTrace();
}

// Perform work as authenticated Subject
...
```

```
// Scope of work complete, logout to remove authentication info
try
{
    lc.logout();
}
catch(LoginException e)
{
    System.out.println("logout failed");
    e.printStackTrace();
}

// A sample MyHandler class
class MyHandler implements CallbackHandler
{
    public void handle(Callback[] callbacks) throws
        IOException, UnsupportedCallbackException
    {
        for (int i = 0; i < callbacks.length; i++)
        {
            if (callbacks[i] instanceof NameCallback)
            {
                NameCallback nc = (NameCallback) callbacks[i];
                nc.setName(username);
            }
            else if (callbacks[i] instanceof PasswordCallback)
            {
                PasswordCallback pc = (PasswordCallback) callbacks[i];
                pc.setPassword(password);
            }
            else
            {
                throw new UnsupportedCallbackException(callbacks[i],
                    "Unrecognized Callback");
            }
        }
    }
}
```

开发者通过创建 `LoginModule` 接口能够实现集成认证技术，从而允许管理员将不同的认证技术插入到应用中。另外，开发者可以将多个 `LoginModule` 连在一起，从而将多种认证技术作为认证过程的组成部分。示例如下：某 `LoginModule` 可以实现基于用户名和密码的认证，而另一个 `LoginModule` 能够借助于硬件服务（比如智能读卡器或生物认证器）完成认证。`LoginModule` 的生命周期由客户创建的 `LoginContext` 对象和 `login` 方法驱动。具体包括如下几个阶段：

- (1) `LoginContext` 使用各个已配置 `LoginModule` 的 `public`、无参数构建器创建相应的

LoginModule 实例。

(2) 调用 LoginModule 的 initialize 方法完成初始化工作。其中，开发者必须保证 Subject 参数不为 null。initialize 的方法名和定义如下：

```
public void initialize(Subject subject, CallbackHandler callbackHandler,  
    Map sharedState, Map options);
```

(3) 调用 login 方法启动认证过程。考虑如下实例，首先提示用户输入用户名和密码，然后验证存储在命名服务，比如 NIS 或 LDAP 中的数据。具体实现可能是借助于智能卡或生物认证服务，或者简单地从操作系统底层抽取用户信息。各个 LoginModule 完成用户身份验证过程可以认为是 JAAS 认证的第一阶段。login 方法名和定义如下：

```
boolean login() throws LoginException;
```

如果上述过程抛出 LoginException 异常，则表示认证失败；如果返回 true，则表明认证成功；如果返回 false，则表明应该忽略该登录模块。

(4) 如果 LoginContext 的所有认证成功，则会调用每个 LoginModule 的 commit 方法。如果第一阶段成功运行 LoginModule，commit 方法将开始进入第二阶段，即将相关 Principal、公共凭证或私有凭证提供给 Subject。如果第一阶段失败，将会删除已经存储的状态，比如用户名或密码。commit 的方法名和定义如下：

```
boolean commit() throws LoginException;
```

如果上述过程抛出 LoginException 异常，则表明 commit 阶段失败；如果返回 true，则表明方法成功；如果返回 false，则表明应该忽略该登录模块。

(5) 如果 LoginContext 的所有认证失败，则调用每个 LoginModule 的 abort 方法。abort 方法删除或销毁由 login 或 initialize 方法创建的认证状态资料。abort 的方法名和定义如下：

```
boolean abort() throws LoginException;
```

如果上述过程抛出 LoginException 异常，则表明 abort 阶段失败；如果返回 true，则表明方法成功；如果返回 false，则表明应该忽略该登录模块。

(6) 在成功登录后，当应用调用 LoginContext 的 logout 方法时，会删除登录过程中保存的认证状态信息。随后，会依次触发各个 LoginModule 的 logout 方法调用。logout 方法删除原先 commit 操作中为 Subject 添加的 Principal 和凭证。一旦执行删除操作，则应该销毁凭证。logout 的方法名和定义如下：

```
boolean logout() throws LoginException;
```

如果上述过程抛出 LoginException 异常，则表明 logout 阶段失败；如果返回 true，则表明方法成功；如果返回 false，则表明应该忽略该登录模块。

当 LoginModule 同用户通信获得认证资料时，它会调用 CallbackHandler 对象。应用程序应该实现 CallbackHandler 接口，并将它传递给 LoginContext。其中，LoginContext 将 CallbackHandler 直接传给 LoginModule。LoginModule 使用 CallbackHandler 既能收集用户输入资料（比如密码或智能卡 PIN 码），也能为用户提供资料（比如用户的状态信息）。借助于应用指定 CallbackHandler，使得底层 LoginModule 能够独立于应用同用户的具体交互。

方式。比如，某 GUI 应用的 CallbackHandler 实现可能是显示窗口以提示用户输入信息。再比如，对于非 GUI 环境的 CallbackHandler 实现（如应用服务器），它可能只是使用应用服务器提供的 API 获得凭证资料。CallbackHandler 接口需要实现如下方法：

```
void handle(Callback[] callbacks)
throws java.io.IOException, UnsupportedCallbackException;
```

接下来，本节来讲述最后一个认证类——Callback 接口。它是一标记（tagging）接口。JAAS 提供了若干默认实现，在列表 8-6 中给出了 NameCallback 和 PasswordCallback 实现。LoginModule 使用 Callback 获得其封装认证机制所要求的资料。在认证的 login 阶段，LoginModule 直接传递 Callback 数组给 CallbackHandler.handle 方法。如果 CallbackHandler 不能解析传入 handle 方法的 Callback 对象，则抛出 UnsupportedCallbackException 异常，从而放弃 login 调用。

8.3 JBoss 安全性模型

类似于 JBoss 架构的其他部分，JBoss 在最底层将安全性定义为接口集合，使得开发者更换安全性实现成为可能。定义 JBoss 服务器安全性层的 3 个基本接口如下：

- org.jboss.security.AuthenticationManager
- org.jboss.security.RealmMapping
- org.jboss.security.SecurityProxy

安全性接口的类图如图 8-3 所示，其中还给出了安全性接口与 EJB 容器架构的关系。

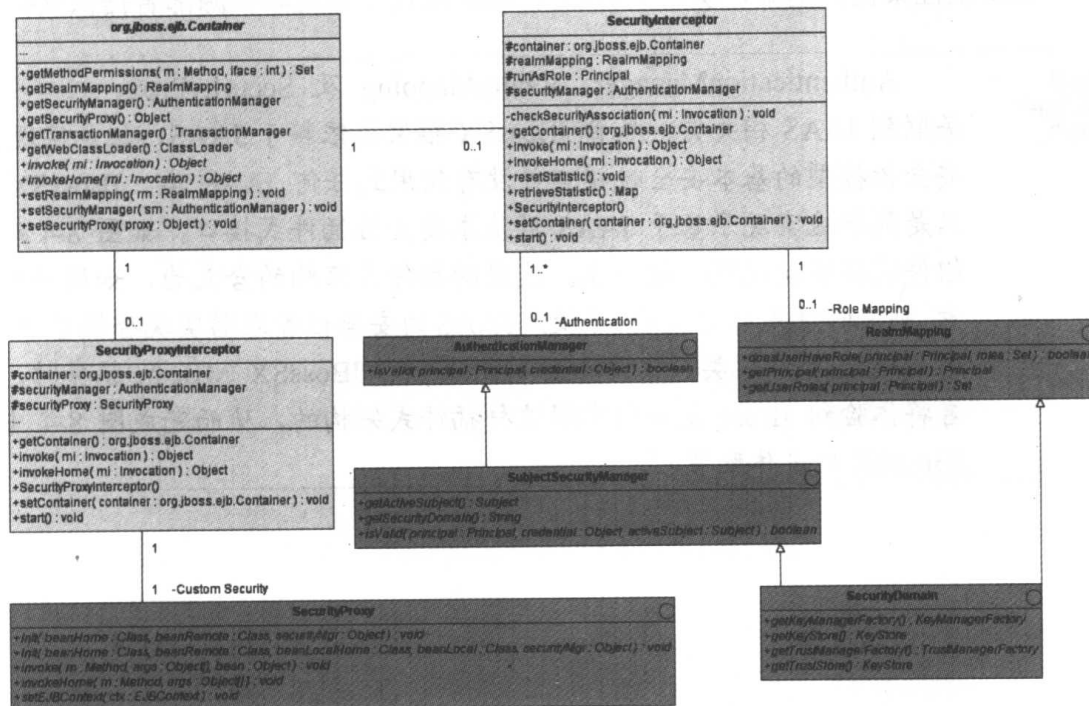


图 8-3 主要的安全性模型接口及其与 JBoss 服务器 EJB 容器元素的关系

其中，图中深色类代表安全性接口，浅色类代表 EJB 容器层。实现 J2EE 安全性模型所需的两个接口是 `org.jboss.security.AuthenticationManager` 和 `org.jboss.security.RealmMapping`。现将图 8-3 显示的安全性接口的作用总结如下：

- **AuthenticationManager**: `AuthenticationManager` 接口负责验证 `Principal` 关联的凭证信息。其中，`Principal` 代表了身份，如用户名、雇员号、社会保险号等等。凭证是身份证明，如密码、会话键、数字签名等等。开发者调用 `isValid` 方法能够判断操作环境中用户身份及其凭证是否是用户身份的有效证明。
- **RealmMapping**: `RealmMapping` 接口负责 `Principal` 和角色映射。其中，`getPrincipal` 方法使用操作环境中的用户身份作为参数，并返回应用域身份。`doesUserHaveRole` 方法判断操作环境中的用户身份是否分配给了应用域中的指定角色。
- **SecurityProxy**: `SecurityProxy` 接口描述用于自定义 `SecurityProxyInterceptor` 插件的需求。其中，`SecurityProxy` 允许外部自定义安全性，即基于方法粒度，以检查 EJB home 和远程接口方法。
- **SubjectSecurityManager**: `SubjectSecurityManager` 是 `AuthenticationManager` 的子接口。它只是简单地添加访问方法，以获得安全性管理器的安全性域名和当前线程的已认证 `Subject`。在 JBoss 的后续版本中，会将其集成到 `SecurityDomain` 接口中。
- **SecurityDomain**: `SecurityDomain` 扩展了 `AuthenticationManager`、`RealmMapping` 及 `SubjectSecurityManager`。它是基于 JAAS `Subject`、`java.security.KeyStore`、`JSSE com.sun.net.ssl.KeyManagerFactory` 及 `com.sun.net.ssl.TrustManagerFactory` 接口的、功能丰富的安全性接口。目前，该接口还在开发中，JBoss 将会把它作为多域安全性架构的基础，从而更好地为 ASP 模式的应用和资源部署提供支持。

注意

`AuthenticationManager`、`RealmMapping` 及 `SecurityProxy` 接口并没有关联到 JAAS 相关类。尽管 JBossSX 框架很依赖于 JAAS，但是实现 J2EE 安全性模型的基本安全性接口并没有使用到任何 JAAS 类。JBossSX 框架只是简单地实现了基于 JAAS 的基本安全性插件式接口。如图 8-4 给出的组件流程图就说明了这一点。这里的插件式架构的含义为，如果开发者乐意，则能够使用自定义的、不使用 JAAS 的安全性管理器实现替换基于 JAAS 的 JBossSX 实现类。本章在后半部分介绍 JBossSX MBean 服务时，开发者将体验到 JBoss 是如何实现这种插件式架构的，从而完成图 8-4 中给出 JBossSX 的具体配置。

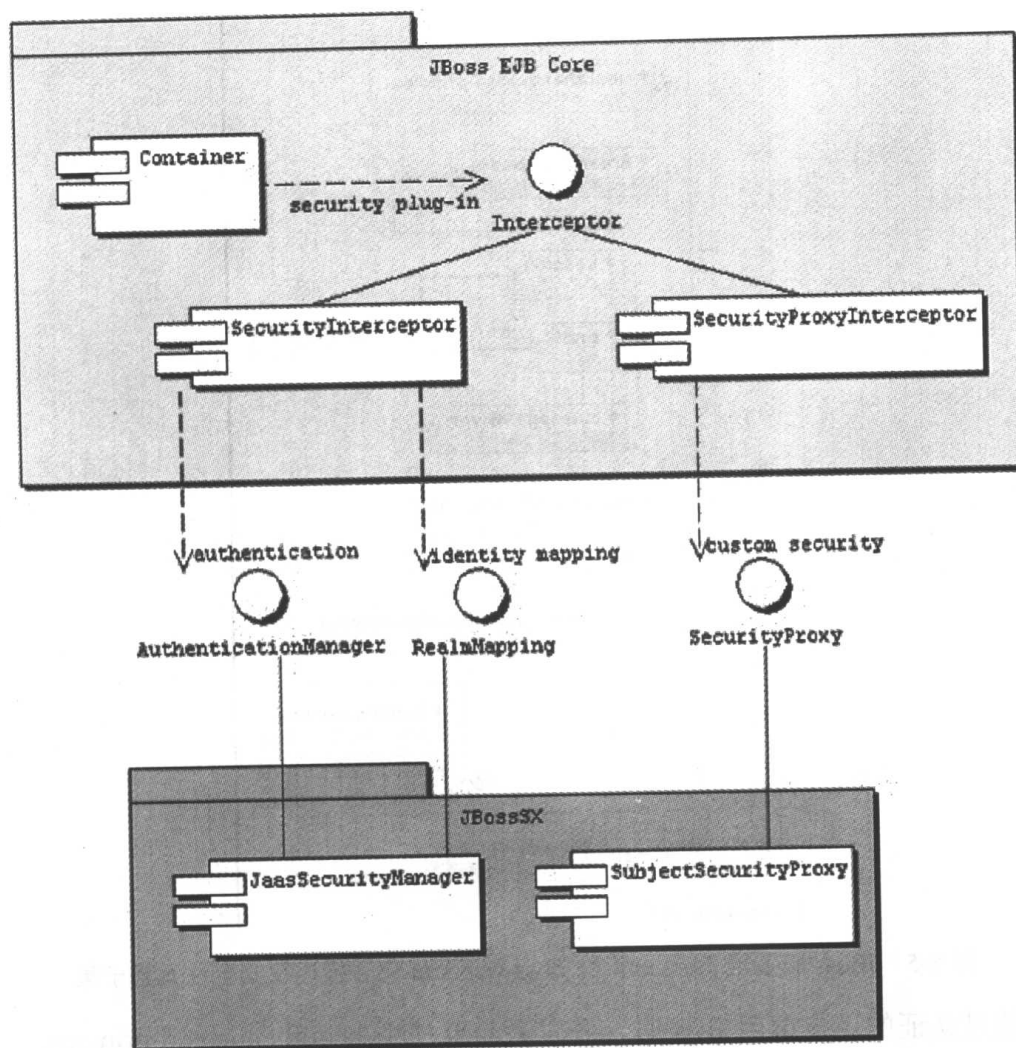


图 8-4 JBossSX 框架实现类和 JBoss 服务器 EJB 容器层之间的关系

再次生效 JBoss 安全性声明

本书在讨论标准 J2EE 安全性模型时，曾提到如何使用 JBoss 服务器特定部署描述符来生效安全性需求。现在，本节来给出其具体配置细节，这也是通用 JBoss 安全性模型的组成部分。图 8-5 给出了 EJB 和 Web 应用中 JBoss 特定部署描述符的安全性相关元素。

security-domain 元素值指定安全性管理器接口实现的 JNDI 名。其中，JBoss 需要使用上述管理器为 EJB 和 Web 容器提供服务。它同时实现了 AuthenticationManager 和 RealmMapping 接口的对象。当开发者将它指定为顶级元素时，实际上是为部署单元中的所有 EJB 指定了生效的安全性域。这是通常的用法，因为在部署单元中混合使用多个安全性管理器使得组件间操作和管理复杂化。

为单个 EJB 指定 security-domain，开发者需要在容器配置级指定 security-domain。同时，各个 EJB 指定的 security-domain 元素将覆盖顶级 security-domain 元素。

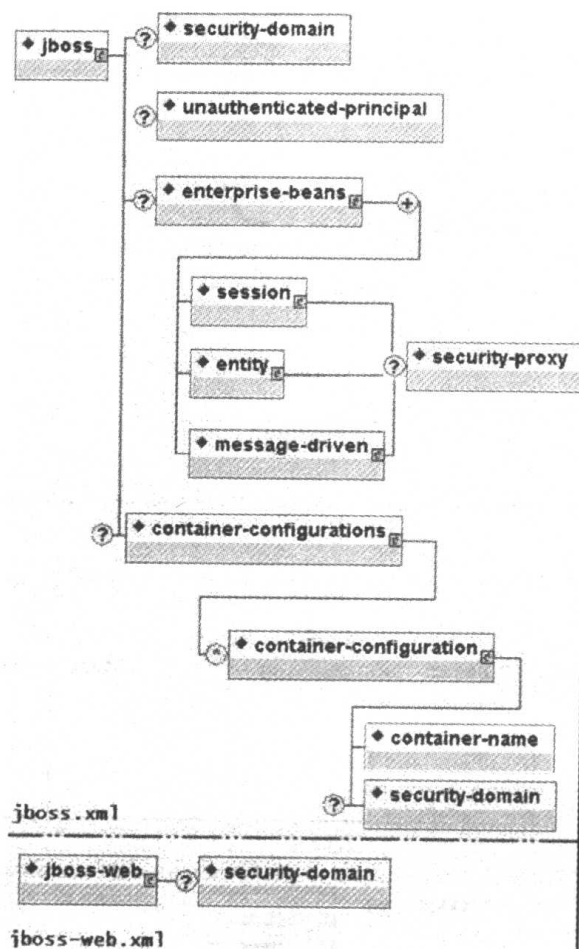


图 8-5 JBoss 服务器 jboss.xml 和 jboss-web.xml 部署描述符安全性元素子集

当未通过认证的用户调用 EJB 时, 开发者需要使用 `unauthenticated-principal` 元素指定 `EJBContext.getUserPrincipal` 方法返回 `Principal` 对象使用的名字。请开发者注意, 它并没有给未认证用户提供任何特殊许可。其主要目的是允许未受保护的 Servlet 和 JSP 页面能够调用未受保护的 EJB, 并允许目标 EJB 使用 `getUserPrincipal` 方法获得调用者的、非 null 值的 `Principal`。这些都是为了遵循 J2EE 规范。

`security-proxy` 元素指定自定义安全性代理实现。其中, 它允许在 EJB 安全性声明模型外部, 对每次请求进行安全性检查, 即不需要将安全性逻辑嵌入到 EJB 实现中。同时, 它可能实现了 `org.jboss.security.SecurityProxy` 接口。或者, 它是没有实现任何通用接口的对象, 即只是在待受保护 EJB 的 home、远程接口、本地 home 或本地接口中实现了方法的对象。如果指定类没有实现 `SecurityProxy` 接口, 则实例必须包裹在 `SecurityProxy` 实现中, 从而将方法调用委派给它。另外, JBossSX 默认情况下使用的元素值(即 `org.jboss.security.SubjectSecurityProxy`), 是实现了 `SecurityProxy` 的实例之一。

接下来, 本节同开发者一起研究一简单实例, 即试验性的、无状态会话 Bean 上下文使用的自定义 `SecurityProxy` 实例。其中, 该自定义 `SecurityProxy` 能够保证, 客户不能够使用长度为 4 个字母的参数来调用企业 Bean 的 `echo` 方法。这种检查如果使用基于角色的安全性, 则不可能实现, 因为这里的安全性上下文是方法参数, 而不是调用者的属性。因此, 不能够定义角色 `FourLetterEchoInvoker`。列表 8-7 给出了自定义 `SecurityProxy` 代码,

完整的源代码请参考 src/main/org/jboss/chap8/ex1 目录。列表 8-8 给出了与其相关的 jboss.xml 描述符，它将 EchoSecurityProxy 配置成 EchoBean 的自定义代理。

列表 8-7 基于参数的安全性约束的自定义 EchoSecurityProxy 实现实例

```
package org.jboss.chap8.ex1;

import java.lang.reflect.Method;
import javax.ejb.EJBContext;

import org.apache.log4j.Category;

import org.jboss.security.SecurityProxy;

/** A simple example of a custom SecurityProxy implementation
    that demonstrates method argument based security checks.

    * @author Scott.Stark@jboss.org
    * @version $Revision:$
    */
public class EchoSecurityProxy implements SecurityProxy
{
    Category log = Category.getInstance(EchoSecurityProxy.class);
    Method echo;

    public void init(Class beanHome, Class beanRemote,
        Object securityMgr)
        throws InstantiationException
    {
        log.debug("init, beanHome="+beanHome
            + ", beanRemote="+beanRemote
            + ", securityMgr="+securityMgr);
        // Get the echo method for equality testing in invoke
        try
        {
            Class[] params = {String.class};
            echo = beanRemote.getDeclaredMethod("echo", params);
        }
        catch (Exception e)
        {
            String msg = "Failed to find an echo(String) method";
            log.error(msg, e);
            throw new InstantiationException(msg);
        }
    }

    public void setEJBContext(EJBContext ctx)
    {

```

```

        log.debug("setEJBContext, ctx="+ctx);
    }

    public void invokeHome(Method m, Object[] args)
        throws SecurityException
    {
        // We don't validate access to home methods
    }

    public void invoke(Method m, Object[] args, Object bean)
        throws SecurityException
    {
        log.debug("invoke, m="+m);
        // Check for the echo method
        if( m.equals(echo) )
        {
            // Validate that the msg arg is not 4 letter word
            String arg = (String) args[0];
            if( arg == null || arg.length() == 4 )
                throw new SecurityException("No 4 letter words");
        }
        // We are not responsible for doing the invoke
    }
}

```

列表 8-8 将 EchoSecurityProxy 配置成 EchoBean 的自定义安全性代理的 jboss.xml 描述符

```

<jboss>
  <security-domain>java:/jaas/other</security-domain>

  <enterprise-beans>
    <session>
      <ejb-name>EchoBean</ejb-name>
      <security-proxy>org.jboss.chap8.ex1.EchoSecurityProxy
    </security-proxy>
    </session>
  </enterprise-beans>
</jboss>

```

EchoSecurityProxy 检查被调用企业 Bean 实例的方法同 EchoSecurityProxy.init 方法装载的 echo(String)是否一致。如果一致，则获得其方法参数，然后将其长度与 4 或 null 作比较。如果等于 4 或者为 null，则显示 SecurityException 异常。当然，这只是人为的应用实例。其中，应用必须根据方法参数值完成安全性检查，这也是常见的应用需求。上述实例演示了如何实现超出标准安全性声明模型范围的自定义安全性与企业 Bean 实现的独立性。安全性专家需要完成安全性需求的规范定义和编码。既然安全性代理层能够独立于企业 Bean 实现，则可以根据具体的部署环境需求更换其安全性实现。

接下来，通过运行客户应用来测试上述自定义代理，即分别通过参数“Hello”和“Four”

调用 EchoBean.echo 方法。具体代码片段如下：

```
public class ExClient
{
    public static void main(String args[]) throws Exception
    {
        Logger log = Logger.getLogger("ExClient");
        log.info("Looking up EchoBean");
        InitialContext iniCtx = new InitialContext();
        Object ref = iniCtx.lookup("EchoBean");
        EchoHome home = (EchoHome) ref;
        Echo echo = home.create();
        log.info("Created Echo");
        log.info("Echo.echo('Hello') = "+echo.echo("Hello"));
        log.info("Echo.echo('Four') = "+echo.echo("Four"));
    }
}
```

其中，第一个调用应该会成功，但第二个会失败，因为“Four”是4个字母的单词。开发者可以在目录 examples 下运行 ant 命令行：

```
[nr@toki examples]$ ant -Dchap=chap8 -Dex=1 run-example
run-example1:
[copy] Copying 1 file to /tmp/jboss-3.2.3/server/default/deploy
[echo] Waiting for 5 seconds for deploy...
[java] [INFO,ExClient] Looking up EchoBean
[java] [INFO,ExClient] Created Echo
[java] [INFO,ExClient] Echo.echo('Hello') = Hello
[java] Exception in thread "main" java.rmi.ServerException: RemoteException occurred in server
thread; nested exception is:
[java] java.rmi.ServerException: RuntimeException; nested exception is:
[java] java.lang.SecurityException: No 4 letter words
...
[java] at org.jboss.chap8.ex1.ExClient.main(ExClient.java:23)
[java] Caused by: java.rmi.ServerException: RuntimeException; nested exception is:
[java] java.lang.SecurityException: No 4 letter words
...
[java] Exception in thread "main"
[java] Java Result: 1
```

上述代码执行结果为，echo('Hello')方法调用成功，这同期待结果相一致。但是，echo('Four')方法调用失败，并产生了相当直观的异常信息，这也和期待结果相吻合。上述给出的输出结果是经过处理的，最重要的一点是，异常的主要部分在于 EchoSecurityProxy 抛出的 SecurityException("No 4 letter words")异常信息，它表明客户应用放弃其尝试的方法调用。

8.4 JBoss 安全性扩展架构

本章在讨论通用 JBoss 安全性层时曾提到，JBossSX 安全性扩展框架实现了安全性层接口，这正是 JBossSX 框架的主要目的。这些实现细节很有意思，因为它提供了大量的自定义工作，以集成到现有的安全性基础框架中。安全性基础框架可能是数据库或 LDAP 服务器，甚至是高级安全性软件的套件。使用 JAAS 框架中的插入式认证模型提供了集成的灵活性，这也是能够集成上述安全性基础框架的重要基础。

JBossSX 框架的核心是 `org.jboss.security.plugins.JaasSecurityManager`，它是 `AuthenticationManager` 和 `RealmMapping` 接口的默认实现。其中，图 8-6 给出了如何借助于组件部署描述符中的 `security-domain` 元素集成 `JaasSecurityManager` 到 EJB 和 Web 容器层。

图 8-6 表明，包含了 EJB 和 Web 内容的企业应用被保护在安全域 `jwdomain` 下。EJB 和 Web 容器的请求拦截器架构包括了安全性拦截器，从而加强了容器的安全性模型。在部署阶段，使用 `jboss.xml` 和 `jboss-web.xml` 描述符中的 `security-domain` 元素值能够获得与容器关联的安全性管理器实例，然后安全性拦截器使用安全性管理器来操作角色。当客户请求受保护组件时，安全性拦截器会把安全性检查委派给与容器关联的安全性管理器实例。

JBossSX `JaasSecurityManager` 实现，即图 8-6 中展示的 `JaasSecurityManager` 组件，使用 `Subject` 实例关联的用户资料进行安全性检查。其中，执行 JAAS 登录模块能够返回 `Subject` 实例，而该 JAAS 登录模块配置在 `security-domain` 元素下。下面将讨论 `JaasSecurityManager` 实现及它是如何使用 JASS。

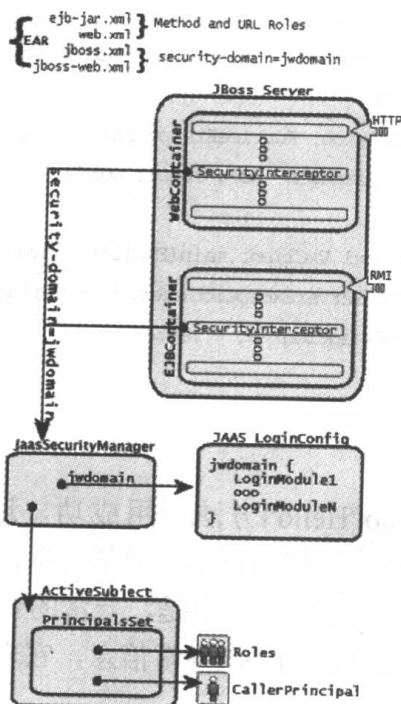


图 8-6 security-domain 组件部署描述符值、组件容器以及 `JaasSecurityManager` 之间的关系

8.4.1 JaasSecurityManager 如何使用 JAAS

JaasSecurityManager 使用 JAAS 包实现了 AuthenticationManager 和 RealmMapping 接口。特别地，它的行为来自于执行分配给它的登录模块实例，而这些登录模块配置在能够匹配分配给 JaasSecurityManager 的安全性域的名字下。登录模块实现了安全性域的 Principal 认证和角色映射。因此，开发者只需要简单地将 JaasSecurityManager 插入到域中的不同登录模块配置里，便能够在不同安全性域中使用 JaasSecurityManager。

为阐述 JaasSecurityManager 中 JAAS 认证过程的详细内容，本节将研究客户调用 EJB Home 方法的具体过程。在此之前，开发者需要先完成如下前提工作：第一，EJB 已经部署在 JBoss 服务器中；第二，使用了 ejb-jar.xml 描述符中的 method-permission 元素保护 EJB Home 接口方法；第三，使用 jboss.xml 描述符中的 security-domain 元素指定了安全性域“jwdomain”。

图 8-7 给出了客户与服务器通信的视图，具体步骤如下。

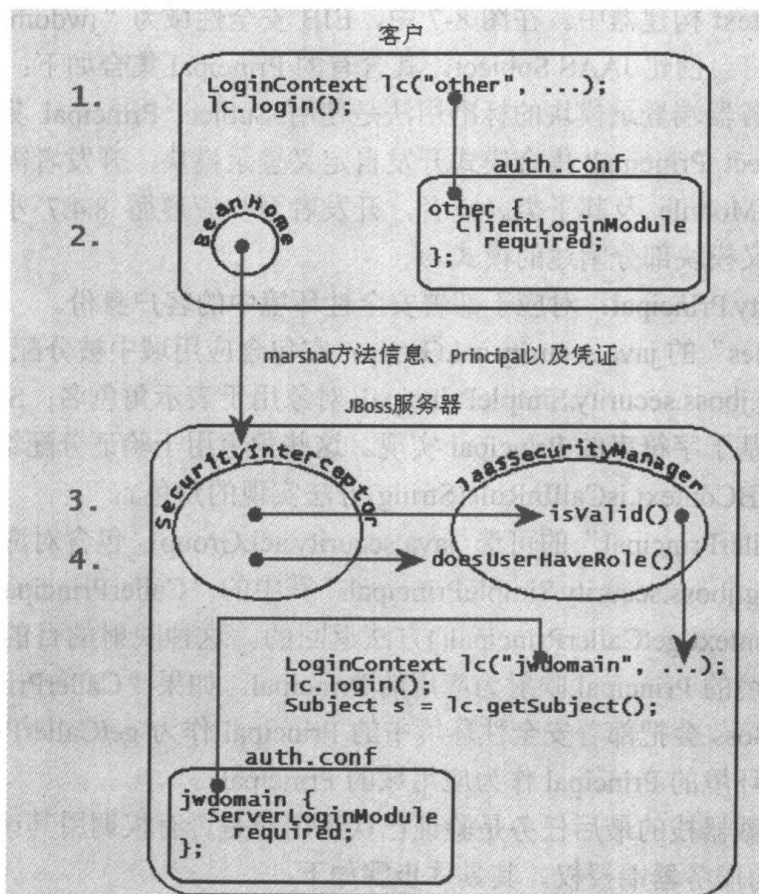


图 8-7 受保护 EJB Home 方法调用的认证和授权过程

步骤

(1) 客户首先完成 JAAS 登录，建立用于认证的 Principal 和凭证，图 8-7 将其称为客

户端登录，这也是 JBoss 中客户建立登录身份的方式。在 3.2.1 节中的“Login InitialContext 工厂实现”给出了借助于 JNDI InitialContext 属性获得登录信息的另一种方式。JAAS 登录将创建 LoginContext 实例，并将配置名传递给它，图 8-7 中配置名为“other”。登录一次所关联的登录 Principal 和凭证能够供随后的所有 EJB 方法使用。请开发者注意，目前暂时还没有认证该用户。客户端登录行为依赖于客户使用的登录模块配置，在图 8-7 中，客户端登录配置入口“other”使用了 ClientLoginModule 模块（org.jboss.security.ClientLoginModule）。这也是 JBoss 服务器的默认客户端模块，它简单地将用户名和密码绑定到 JBoss EJB 调用层，以便用于后续认证操作。客户端不能够认证客户身份。

(2) 随后客户获得 EJB Home 接口，并试图创建企业 Bean 实例。图 8-7 将其称为 home 方法调用，它将触发 Home 接口方法调用发送到 JBoss 服务器。该调用包含了客户传递的方法参数及上述步骤完成的客户端 JAAS 登录获得的用户身份和凭证。

(3) 在服务器端，安全性拦截器首先要求认证触发上述调用的用户。这时该用户在客户端已经完成了 JAAS 登录。

(4) 保护 EJB 的安全域决定了具体使用的登录模块。安全域名被作为登录配置入口名传递到 LoginContext 构建器中。在图 8-7 中，EJB 安全性域为“jwddomain”。如果 JAAS 登录认证该用户，将会创建 JAAS Subject。其含有的 Principal 集合如下：

(JBossSX 服务器端登录模块的标准用法是使用 Subject Principal 集合模式。为确保正确地使用了 Subject Principal 集合模式开发自定义登录模块，开发者需要实现 JBossSX AbstractServerLoginModule 及其子类。或者，开发者至少应遵循 8.4.7 小节“开发自定义登录模块”中自定义模块部分阐述的模式。)

- java.security.Principal：对应于部署安全性环境中的客户身份。
- 称为“Roles”的 java.security.acl.Group：它包含应用域中被分配给用户的角色名。其中，org.jboss.security.SimplePrincipal 对象用于表示角色名；SimplePrincipal 是简单的、基于字符串的 Principal 实现。这些角色用于验证分配给 ejb-jar.xml 中的方法和 EJBContext.isCallInRole(String)方法实现的角色。
- 称为“CallerPrincipal”的可选 java.security.acl.Group：包含对应于应用域调用者身份的 org.jboss.security.SimplePrincipal。其中的“CallerPrincipal”单独组成员是由 EJBContext.getCallerPrincipal()方法返回的。这种映射的目的在于能够将操作安全性环境的 Principal 映射为应用的 Principal。如果“CallerPrincipal”映射不存在，则 JBoss 会把部署安全性环境中的 Principal 作为 getCallerPrincipal 方法值，即将操作环境的 Principal 作为应用域的 Principal。
- 安全性拦截器栈的最后任务是验证已认证用户是否有权调用其请求的方法。在图 8-7 中称为服务器端授权。其具体步骤如下：
 - 获得角色名，以便访问 EJB 容器中的 EJB 方法。其中，这些角色名是通过 ejb-jar.xml 描述符中包含了被调用 EJB 方法的所有 method-permission 元素的 role-name 子元素指定的。
 - 如果没有为这些 EJB 方法指定角色，或者方法被指定在 exclude-list 元素中，则不允许应用客户访问请求的方法。否则，安全性拦截器将会调用 JaasSecurityManager.doesUserHaveRole (Principal,Set) 方法，从而能够判断

调用者是否处于其包含的角色名中。doesUserHaveRole 方法实现将迭代角色名，并检查已认证用户的 Subject 的“Roles”组，从而能够判断其含有的 SimplePrincipal 是否被指定了要求的角色名。如果角色名是“Roles”组的成员，则允许客户访问。如果角色名不是其成员，则 EJB 容器不允许它访问被调用的 EJB 方法。

- 如果开发者为 EJB 配置了自定义安全性代理，则方法调用操作将委派给它。如果安全性代理不允许调用者访问它，则会抛出 java.lang.SecurityException 异常。如果安全性代理没有抛出 SecurityException 异常，则允许访问 EJB 方法，并将方法调用传递给下一个容器拦截器。请开发者注意，SecurityProxyInterceptor 处理该项检查，但在图 8-7 中并没有给出。

由于安全性信息被处理为请求包含的无状态属性，即每个请求都必须提供和验证安全性信息，因此每个受保护 EJB 方法调用和 Web 内容访问都需要对调用者进行认证和授权。如果 JAAS 登录涉及到客户-服务器通信，则是很耗费资源的操作。据此，JaasSecurityManager 引入了认证缓存，即将成功登录中的 Principal 和凭证信息保存在缓存中。开发者可以将认证缓存实例配置成 JaasSecurityManager 的组成部分。在讨论相关 MBean 服务时，开发者能够看到认证缓存的具体配置过程。如果不存在任何用户定义的缓存，服务器将使用默认缓存，以维护凭证信息。与此同时，开发者可以设置默认缓存的作用时间。

8.4.2 JaasSecurityManagerService MBean

JaasSecurityManagerService MBean 服务用于管理安全性管理器。尽管该 MBean 服务名以“Jaas”开头，但它处理的安全性管理器实现可以不使用 JAAS。

该名字源于如下事实：默认安全性管理器实现是 JaasSecurityManager。JaasSecurityManagerService 的主要作用是外部配置（externalize）安全性管理器实现。通过提供实现了 AuthenticationManager 和 RealmMapping 接口的其他安全性管理器，开发者能够更换其默认实现。当然，这里只是多提供一种供开发者选择的安全性管理器实现，因为在默认情况下 JBoss 将使用 JaasSecurityManager 实现。

另外，JaasSecurityManagerService MBean 的第二个基本作用是提供 JNDI javax.naming.spi.ObjectFactory 实现，即通过简单的、不需要代码的 JNDI 名管理，从而能够实现到安全性管理器的映射。借助于 security-domain 部署描述符元素能够指定安全性管理器实现的 JNDI 名，这同时也生效了安全性。当开发者指定 JNDI 名时，需要给出对象绑定的位置。为简化安全性管理器绑定的 JNDI 名设置，JaasSecurityManagerService 将安全性管理器实例关联到 JNDI “java:/jaas”名下的命名系统，该命名系统被 JaasSecurityManagerService 引用为 JNDI ObjectFactory。开发者可以为元素 security-domain 取值使用“java:/jaas/XYZ”形式的命名规范。同时，服务器将创建“XYZ”安全性域所需的安全性管理器实例。初次查找“java:/jaas/XYZ”绑定时，JBoss 会为“XYZ”域创建安全性管理器，即使用带有安全性域名参数的构建器创建 SecurityManagerClassName 属性指定的类实例。比如，请开发者考虑如下容器安全性配置片段。

```
<jboss>
  <!-- Configure all containers to be secured under the
```



```
"hades" security domain -->
<security-domain>java:/jaas/hades</security-domain>
...
</jboss>
```

当查找“java:/jaas/hades”名时，服务器将会返回一关联安全性域“hades”的安全性管理器实例。其中，该安全性管理器实现了 `AuthenticationManager` 和 `RealmMapping` 接口，它同时也是 `JaasSecurityManagerService` MBean 中的 `SecurityManagerClassName` 属性指定的类型。

标准 JBoss 发布版在默认情况下，使用了 `JaasSecurityManagerService` MBean 配置，开发者通常只需要使用该默认配置即可。`JaasSecurityManagerService` MBean 的可配置属性如下：

- **SecurityManagerClassName:** `SecurityManagerClassName` 属性指定安全性管理器实现的类名。其中，该实现类必须同时支持 `org.jboss.security.AuthenticationManager` 和 `org.jboss.security.RealmMapping` 接口。如果开发者没有指定该属性，在默认情况下其取值是基于 JAAS 的 `org.jboss.security.plugins.JaasSecurityManager`。
- **CallbackHandlerClassName:** `CallbackHandlerClassName` 属性指定供 `JaasSecurityManager` 使用的 `javax.security.auth.callback.CallbackHandler` 实现类名。如果默认 `CallbackHandler` 实现，即 `org.jboss.security.auth.callback.SecurityAssociationHandler`，不能够满足开发者的要求，那么开发者可以更换 `JaasSecurityManager` 使用的处理器。除非开发者已经深入掌握了这项功能，否则不要修改它，因为这是一个技术难度很高的配置。
- **SecurityProxyFactoryClassName:** `SecurityProxyFactoryClassName` 属性指定 `org.jboss.security.SecurityProxyFactory` 实现的类名。如果开发者没有指定其值，则默认值为 `org.jboss.security.SubjectSecurityProxyFactory`。
- **AuthenticationCacheJndiName:** `AuthenticationCacheJndiName` 属性指定安全性凭证缓存策略的位置。起初它被看做是能够根据各个安全性域，具有返回 `CachePolicy` 实例能力的 `ObjectFactory` 位置。当查找安全性域的 `CachePolicy` 时，服务器需要将安全性域名追加到该位置后面。如果该追加步骤失败，则会将该位置看成所有安全性域的 `CachePolicy`。在默认情况下，服务器使用定时缓存策略。
- **DefaultCacheTimeout:** `DefaultCacheTimeout` 属性指定默认定时缓存策略的超时时间（单位：秒）。其默认值为 1 800 秒（30 分钟）。当然，具体取值需要权衡如下几方面：认证操作的频度、凭证信息和安全性信息存储源可能会出现不同步的时间长度。如果开发者不打算缓存安全性凭证，则将该属性取值设置为 0。如果 `AuthenticationCacheJndiName` 属性不是默认取值，则该属性无效。
- **DefaultCacheResolution:** `DefaultCacheResolution` 属性指定默认定时缓存策略精度（单位：秒）。它用于控制更新缓存的当前时间戳的时间间隔。其值应小于 `DefaultCacheTimeout` 取值，否则没有意义，默认值为 60 秒（1 分钟）。如果 `AuthenticationCacheJndiName` 属性不是默认取值，则该属性无效。

另外，`JaasSecurityManagerService` 还提供了很多的有用操作。其具体操作包括：运行

时刷新（flushing）任何安全性域认证缓存、获得安全性域认证缓存中的活动用户列表及获得安全性管理器接口的任何方法。

当更新了底层存储源且用户打算立即使用存储源状态时，刷新安全性域认证缓存将会丢弃所有缓存的凭证。JaasSecurityManagerService MBean 中，对应的操作方法名和定义如下：

```
public void flushAuthenticationCache(String securityDomain);
```

通过程序能够调用它，源代码如下：

```
MBeanServer server = ...;
String jaasMgrName = "jboss.security:service=JaasSecurityManager";
ObjectName jaasMgr = new ObjectName(jaasMgrName);
Object[] params = {domainName};
String[] signature = {"java.lang.String"};
server.invoke(jaasMgr, "flushAuthenticationCache", params, signature);
```

获得活动用户列表的操作给出了安全性域认证缓存中当前没有过期的 Principal 键的快照。JaasSecurityManagerService MBean 中对应的操作方法名和定义如下：

```
public List getAuthenticationCachePrincipals(String securityDomain);
```

通过程序能够调用它，源代码如下：

```
MBeanServer server = ...;
String jaasMgrName = "jboss.security:service=JaasSecurityManager";
ObjectName jaasMgr = new ObjectName(jaasMgrName);
Object[] params = {domainName};
String[] signature = {"java.lang.String"};
List users = (List) server.invoke(jaasMgr,
    "getAuthenticationCachePrincipals", params, signature);
```

JBoss 3.0.5 添加了如下安全性管理器访问方法：

```
public boolean isValid(String securityDomain, Principal principal,
    Object credential);
public Principal getPrincipal(String securityDomain, Principal principal);
public boolean doesUserHaveRole(String securityDomain, Principal principal,
    Set roles);
public Set getUserRoles(String securityDomain, Principal principal);
```

它们能够访问 securityDomain 参数指定的安全域对应的 AuthenticationManager 和 RealmMapping 接口的方法。

8.4.3 扩展 JaasSecurityManager, JaasSecurityDomain MBean

org.jboss.security.plugins.JaasSecurityDomain 扩展了 JaasSecurityManager。为支持 SSL 和其他加密场合，JaasSecurityDomain 添加了 KeyStore、JSSE KeyManagerFactory 以及

TrustManagerFactory。JaasSecurityDomain MBean 的可配置属性如下：

- **KeyStoreType**: KeyStoreType 属性指定 KeyStore 实现的类型。java.security.KeyStore.getInstance(String type) 工厂方法中的 type 参数就是该属性值。
- **KeyStoreURL**: KeyStoreURL 属性指定 KeyStore 数据库位置的 URL。开发者使用 URL.openStream() 方法能够获得 java.io.InputStream，以装载 KeyStore 实例的内容。
- **KeyStorePass**: KeyStorePass 属性指定 KeyStore 数据库内容关联的密码。当开发者使用 KeyStore.load(InputStream, char[]) 方法从 KeyStoreURL 内容装入 KeyStore 实例时，需要使用该属性。如果开发者没有指定该属性，服务器将使用 null 密码，并且不会检查数据库集成性。
- **LoadSunJSSEProvider**: LoadSunJSSEProvider 属性为标志位。它表明应用启动时是否装载 Sun 提供的安全性供应商，即 com.sun.net.ssl.internal.ssl.Provider。当需要使用 Sun JSSE jar 文件，但开发者没有将它们安装为 JDK 1.3 的扩展，则需要该属性。如果使用 JDK 1.4 或其他 JSSE 供应商，则需要将该属性值设置成 false。默认值为，true。
- **ManagerServiceName**: ManagerServiceName 属性指定安全性管理器服务 MBean 的 JMX ObjectName，用于将 JaasSecurityDomain 注册成 JNDI “java:/jaas/<domain>” 下的安全性管理器。其中，<domain> 指传递给该 MBean 构建器的名字，ManagerServiceName 属性的默认值为 “jboss.security:service=JaasSecurityManager”。

8.4.4 基于 XML 的 JAAS 登录配置 MBean

JBoss 3.x 使用 javax.security.auth.login.Configuration 类的自定义实现，即由 org.jboss.security.auth.login.XMLLoginConfig Mean 提供。该配置实现使用 XML 格式，其遵循的 DTD 如图 8-8 所示。

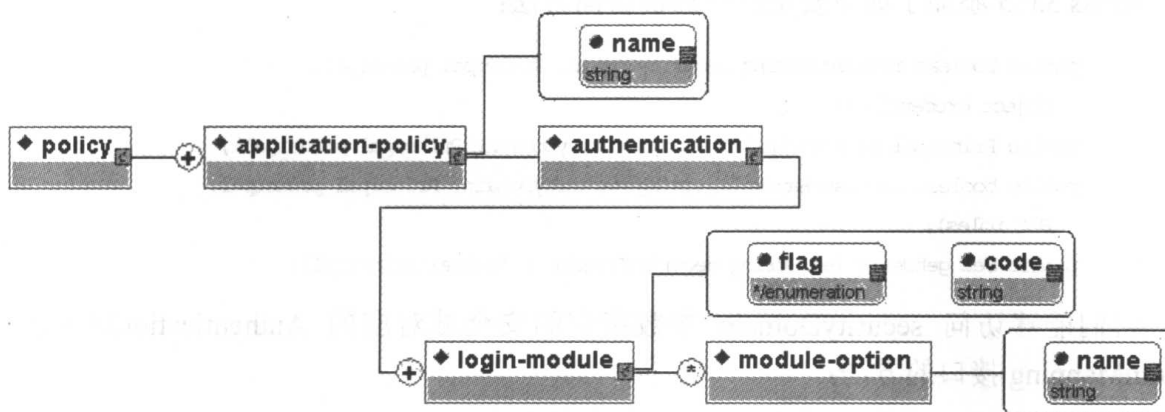


图 8-8 XMLLoginConfig DTD

application-policy 的 name 属性是登录配置名。它对应于 jboss.xml 和 jboss-web.xml 部署描述符 security-domain 元素值中前缀 “java:/jaas” 之后的内容。login-module 元素的 code 属性指定登录模块实现的类名。flag 属性控制认证栈的总体行为，其取值范围及含义如下：

- **required:** 要求 LoginModule 必须成功。不论成功或失败, 认证仍然继续传往 LoginModule 列表处理。
- **requisite:** 要求 LoginModule 必须成功。如果成功, 认证继续传往 LoginModule 列表处理; 如果失败, 控制权立即返还给应用 (认证不再传往 LoginModule 列表)。
- **sufficient:** 对 LoginModule 成功与否不作要求。如果成功, 控制权立即返还给应用 (认证不再传往 LoginModule 列表); 如果失败, 认证继续传往 LoginModule 列表处理。
- **optional:** 对 LoginModule 成功与否不作要求。不论成功或失败, 认证仍然继续传往 LoginModule 列表处理。

login-module 元素可能指定 0 个或多个 module-option 子元素。在初始化登录模块时, 需要使用 module-option 定义的名字/值字符串对。其中, name 属性指定可选名, module-option 元素内容提供值。登录配置实例如列表 8-9 所示。

列表 8-9 适合于 XMLLoginConfig 的登录模块配置实例

```
<policy>
  <application-policy name="srp-test">
    <authentication>
      <login-module code="org.jboss.security.srp.jaas.SRPCacheLoginModule"
        flag="required">
        <module-option name="cacheJndiName">srp-test/AuthenticationCache
        </module-option>
      </login-module>

      <login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule"
        flag="required">
        <module-option name="password-stacking">useFirstPass</module-option>
      </login-module>
    </authentication>
  </application-policy>
</policy>
```

XMLLoginConfig MBean 支持下列属性:

- **ConfigURL:** ConfigURL 属性指定启动时该 MBean 装载 XML 登录配置文件的 URL, 它必须是有效的 URL 字符串表示。
- **ConfigResource:** ConfigResource 属性指定启动时该 MBean 装载 XML 登录配置文件的资源名。在使用线程上下文类装载器定位 URL 时, 该名字被当成是该 URL 的类路径资源。
- **ValidateDTD:** 标志位, 表示是否针对 DTD 验证 XML 配置文件的有效性。默认值为 true。

该 MBean 也支持下列操作, 从而能够在运行时动态扩展登录配置。需要注意的是, 当 JBoss 开发者试图改变登录配置时运行了安全管理器, 则需要添加 javax.security.auth.AuthPermission("refreshLoginConfiguration")。其中, 本书提供的 org.jboss.chap8.service.

SecurityConfig 服务演示了如何动态地添加和删除具体的部署安全性配置。

- `void addAppConfig(String appName,AppConfigurationEntry[] entries)`: 它添加指定的登录模块配置栈到当前配置“appName”下，这将替代“appName”下的任何现存入口。
- `void removeAppConfig(String appName)`: 它删除“appName”下注册的登录模块配置。
- `String[] loadConfig(URL configURL) throws Exception`: 它将从以 XML 或遗留 Sun 登录配置文件形式表示的 URL 中装入一个或多个登录配置。请开发者注意，必须添加所有的登录配置，或者干脆不添加任何登录配置。它返回已添加的登录配置名。
- `void removeConfigs(String[] appNames)`: 它删除“appNames”指定的登录配置。
- `String displayAppConfig(String appName)`: 该操作显示指定配置（如果存在）的简单字符串表示。

8.4.5 JAAS 登录配置管理 MBean

`org.jboss.security.plugins.SecurityConfig` MBean 需要负责安装自定义 `javax.security.auth.login.Configuration` 实现。`SecurityConfig` MBean 的可配置属性如下：

- **LoginConfig**: `LoginConfig` 属性指定提供默认 JAAS 登录配置 MBean 的 JMX ObjectName。当启动 `SecurityConfig` MBean 服务时，它会调用 `getConfiguration(Configuration currentConfig)` 操作以查询 `javax.security.auth.login.Configuration`。如果没有指定 `LoginConfig` 属性，则默认使用 Sun 提供的 `Configuration` 实现，javadoc 中提供了 `Configuration` 类的详细描述。

除了用于安装自定义 JAAS 登录配置实现外，`SecurityConfig` MBean 服务还能够在运行时将这些配置连在一起，形成登录配置栈，从而能够实现登录配置的出栈和压栈任务。安全性单元测试需要使用该特性将自定义登录配置安装到 JBoss 默认发布版中。添加（压栈）新配置的操作如下：

```
public void pushLoginConfig(String objectName) throws JMXException,  
MalformedObjectNameException;
```

类似于 `LoginConfig` 属性，`objectName` 参数指定 MBean。删除当前登录配置操作如下：

```
public void popLoginConfig() throws JMXException;
```

8.4.6 使用和开发 JBossSX 登录模块

`JaasSecurityManager` 实现允许开发者使用 JAAS 登录模块配置完成认证机制的完整自定义工作。通过定义登录模块配置入口（对应于保护 J2EE 组件访问的安全性域名），开发者可以实现认证机制和集成实现。

JBossSX 框架绑定了适合于集成标准的安全性设施存储协议，比如 LDAP 和 JDBC 的

若干登录模块。它也包括标准基类实现，以加强 LoginModule 使用 Subject 模式。在 8.4.7 小节“开发自定义登录模块”中有这方面的阐述。如果没有合适的绑定登录模块，开发者则可以开发自定义认证协议，并将它们集成到 JBoss 服务器中。本节内容首先讨论绑定的登录模块和各自的配置，然后讨论如何开发供 JBoss 使用的自定义 LoginModule 实现。

1. org.jboss.security.auth.spi.IdentityLoginModule

IdentityLoginModule 是简单登录模块，它将模块选项中指定的 Principal 和认证 Subject 关联在一起。它使用“principal”选项指定的名字创建 SimplePrincipal 实例。尽管它并不能够满足产品强度认证条件，但还是可以用于开发环境中，比如当开发者需要测试指定 Principal 及其角色的安全性时。

IdentityLoginModule 具体支持的登录模块配置选项如下：

- **principal=string**: SimplePrincipal 使用的名字，供认证所有用户使用。如果没有指定 principal 选项，则 principal 默认值为“guest”。
- **roles=string-list**: 分配给用户 Principal 的角色名。它是用逗号隔开的角色名列表。
- **password-stacking=useFirstPass**: 当设置了 password-stacking 选项时，IdentityLoginModule 首先会在登录模块共享状态 Map 中寻找属性名“javax.security.auth.login.name”下的共享用户名。如果找到，则将其作为 Principal 名。如果没有，则将 IdentityLoginModule 设置的 Principal 名存储在属性名“javax.security.auth.login.name”下。

下面给出了 Sun 格式的登录配置入口。其中，Principal 为“jduke”、角色名为“TheDuke”和“AnimatedCharacter”。

```
testIdentity {  
    org.jboss.security.auth.spi.IdentityLoginModule required  
    principal=jduke  
    roles=TheDuke,AnimatedCharater;  
};
```

对应 XMLLoginConfig 格式为：

```
<policy>  
  <application-policy name="testIdentity">  
    <authentication>  
      <login-module code="org.jboss.security.auth.spi.IdentityLoginModule"  
        flag="required">  
        <module-option name="principal">jduke</module-option>  
        <module-option name="roles">TheDuke,AnimatedCharater</module-option>  
      </login-module>  
    </authentication>  
  </application-policy>  
</policy>
```

要将其添加到 JBoss 服务器 default 配置文件集合的登录配置中，需要修改 conf/default/auth.conf 文件。

2. org.jboss.security.auth.spi.UsersRolesLoginModule

UsersRolesLoginModule 是另一种简单登录模块。它支持多用户和用户角色，并使用了两个 Java 属性格式的文本文件。其中，用户名到密码映射文件是 `users.properties`，用户名到角色映射文件是 `roles.properties`。在模块的初始化阶段，线程上下文类装载器会使用 `initialize` 方法装载这两个属性文件。这意味着开发者可以将这些文件放置在 J2EE 部署 jar 包、JBoss 配置目录、JBoss 服务器的任何目录或系统类路径中。UsersRolesLoginModule 的主要目的在于使开发者能够很容易使用部署在应用中的属性文件测试多用户和角色的安全性设置。

`users.properties` 文件使用 “`username=password`” 格式，从而在单独行标识各个用户，比如：

```
username1=password1
username2=password2
...
```

`roles.properties` 文件使用 “`username=role1,role2,...`” 格式，并带有可选组名。比如：

```
username1=role1,role2,...
username1.RoleGroup1=role3,role4,...
username2=role1,role3,...
```

“`username.XXX`” 形式的属性名将用户名角色分配给特定角色组，其中 `XXX` 部分为组名。实际上，“`username=...`” 是 “`username.Roles=...`” 的缩写，这里的 “Roles” 组名是 `JaasSecurityManager` 期望的标准名，它定义了用户许可。

下面给出了 `jduke` 用户名的等效表达：

```
jduke=TheDuke,AnimatedCharacter
jduke.Roles=TheDuke,AnimatedCharacter
```

UsersRolesLoginModule 支持的登录模块配置选项如下：

- **unauthenticationIdentity=name:** 定义 Principal 名，即分配给没有包含任何认证信息的客户请求。它能够用于如下场合，即未受保护的 Servlet 调用不要求特定角色的 EJB 方法。由于这种 Principal 没有相关角色，所以该 Principal 只能访问未受保护的 EJB 或使用了 unchecked 许可约束的 EJB 方法。
- **password-stacking=useFirstPass:** 当设置了 password-stacking 选项时，UsersRoles LoginModule 首先会在登录模块共享状态 Map 中寻找属性名 “`javax.security.auth.login.name`” 和 “`javax.security.auth.login.password`” 下的共享用户名和密码。如果找到，则将其作为 Principal 名和密码；如果没找到，则将 UsersRolesLoginModule 设置的 Principal 名和密码分别存储在属性名 “`javax.security.auth.login.name`” 和 “`javax.security.auth.login.password`” 下。
- **hashAlgorithm=String:** `java.security.MessageDigest` 算法名，用于 HASH 加密。在默认情况下，服务器并没有给出该选项，因此开发者必须给出该选项以生效 HASH 加密。当开发者指定了 hashAlgorithm 时，服务器首先会加密 CallbackHandler

返回的明文密码,然后将加密后的明文密码传入到 UsernamePasswordLoginModule.validatePassword 方法中。它是以 inputPassword 参数形式传递的,开发者还须加密存储在 users.properties 文件中的密码。

- **hashEncoding=base64|hex**: 已加密密码的字符串格式,或者为“base64”,或者为“hex”。默认值为 base64。
- **hashCharset=String**: 用于转换明文密码到字节数组的编码(encoding)。默认值为平台默认编码。
- **usersProperties=String**: 为 JBoss 2.4.5 新增属性,用于指定包含用户名到密码映射的属性资源名。默认值为 users.properties。
- **rolesProperties=String**: JBoss 2.4.5 新增属性,用于指定包含用户名到角色映射的属性资源名。默认值为 roles.properties。

下面给出了 Sun 遗留登录配置入口格式实例。其中该实例将未认证用户的 Principal 指定为“nobody”。另外,usersb64.properties 文件的编码为 base64,加密算法为 MD5。

```
testUsersRoles {
    org.jboss.security.auth.spi.UsersRolesLoginModule required
    usersProperties=usersb64.properties
    hashAlgorithm=MD5
    hashEncoding=base64
    unauthenticatedIdentity=nobody
;
};
```

对应的 XMLLoginConfig 格式:

```
<policy>
  <application-policy name="testUsersRoles">
    <authentication>
      <login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule"
        flag="required">
        <module-option name="usersProperties">usersb64.properties
        </module-option>
        <module-option name="hashAlgorithm">MD5</module-option>
        <module-option name="hashEncoding">base64</module-option>
        <module-option name="unauthenticatedIdentity">nobody</module-option>
      </login-module>
    </authentication>
  </application-policy>
</policy>
```

3. org.jboss.security.auth.spi.LdapLoginModule

LdapLoginModule 实现了 LoginModule,它使用登录模块配置提供的 JNDI 登录选项完成基于 LDAP 服务器的认证。如果用户名和凭证信息存储在 LDAP 服务器中,并允许借助于 JNDI LDAP 供应商访问它们,开发者则可以使用 LdapLoginModule。

配置选项中提供的 LDAP 连接信息首先将传递到环境对象中,然后使用它创建 JNDI

InitialContext。标准的 LDAP JNDI 属性如下：

- **java.naming.factory.initial**: InitialContextFactory 实现的类名。默认值为 Sun LDAP 供应商实现，即 `com.sun.jndi.ldap.LdapCtxFactory`。
- **java.naming.provider.url**: LDAP 服务器的 LDAP URL。
- **java.naming.security.authentication**: 使用的安全等级，默认值为，“simple”。
- **java.naming.security.protocol**: 实现保护访问而使用的传输协议，比如 SSL。
- **java.naming.security.principal**: 用于认证调用者的 Principal。通过下文阐述的其他属性能够创建它。
- **java.naming.security.credentials**: 该属性值取决于认证方案。比如，HASH 加密密码、明文密码、密钥、安全证书等等。

LdapLoginModule 登录模块支持的配置选项如下：

- **principalDNPrefix=String**: 添加用户名前缀，用于区分用户名。请参考下文。
- **principalDNSuffix=String**: 添加用户名后缀，用于区分用户名。如果需要输入用户名，但不想麻烦用户同时输入完整的区别名（distinguished name），则需要使用这两个选项。其中：

```
String userDN = principalDNPrefix + username + principalDNSuffix;
```

- **useObjectCredential=true/false**: 指定获得凭证的方式，即或者使用 JAAS PasswordCallback 获得凭证的 `char[]` 密码表示，或者使用 `Callback org.jboss.security.auth.callback.ObjectCallback` 类型获得凭证的透明对象表示。如果开发者使用该属性，则能够将非 `char[]` 形式表示的凭证资料传递给 LDAP 服务器。
- **rolesCtxDN=String**: 固定的区别名，用于搜索用户角色的上下文。
- **userRolesCtxDNAttributeName=String**: 用户对象的属性名，该用户对象含有完整区别名。它和 `rolesCtxDN` 的区别在于，它为搜索用户角色提供的上下文对于每个用户而言，可能是惟一的。
- **roleAttributeID=String**: 含有用户角色的属性名。如果未指定，则默认值为，“roles”。
- **roleAttributeIsDN=String**: 标志位，表明 `roleAttributeID` 是否含有角色对象或角色名的完整区别名。如果选项取值为 `false`，则将 `roleAttributeID` 值赋给角色名；如果选项取值为 `true`，则角色对象的区别名是 `role` 属性值。其中，角色名来自 `roleNameAttributeID` 选项值。在特定目录方案（比如，MS 活动目录）中，用户对象的 `role` 属性被存储为区别名，而不是简单的用户名。对于这种场合，应该将 `roleAttributeIsDN` 选项设置为 `true`，`roleAttributeIsDN` 选项的默认值为 `false`。
- **roleNameAttributeID=String**: 上下文中指向包含角色名的 `roleCtxDN` 区别名值的属性名。如果 `roleAttributeIsDN` 属性值为 `true`，则使用该属性寻找角色对象的 `name` 属性。默认值为“group”。
- **uidAttributeID=String**: 指那些含有用户角色对象的属性名，它用于定位用户角

色。默认值为“uid”。

- **matchOnUserDN=true|false**: 标志位, 即搜索用户角色是否应该匹配用户的完整区别名。如果为 false, 则只使用用户名匹配 uidAttributeName 属性值; 如果为 true, 则使用完整区别名。
- **unauthenticatedIdentity=String**: 定义 Principal 名, 即分配给没有包含任何认证信息的客户请求。该行为继承于 UsernamePasswordLoginModule 超类。
- **password-stacking=useFirstPass**: 当开发者设置了 password-stacking 选项时, LdapLoginModule 首先会在登录模块共享状态 Map 中分别寻找属性名“javax.security.auth.login.name”和“javax.security.auth.login.password”下的共享用户名和密码。如果找到, 则将其作为 Principal 名和密码; 如果没找到, 则将 LdapLoginModule 设置的 Principal 名和密码存储在属性名“javax.security.auth.login.name”和“javax.security.auth.login.password”下。
- **hashAlgorithm=String**: java.security.MessageDigest 算法名, 用于 HASH 加密。在默认情况下, 服务器并没有给出该选项, 因此必须给出该选项以生效 HASH 加密。当开发者指定了 hashAlgorithm 时, 服务器首先会加密 CallbackHandler 返回的明文密码, 然后将加密后的明文密码传入到 UsernamePasswordLoginModule.validatePassword 方法中。其中, 上述明文密码是作为 inputPassword 参数传递的。另外, 开发者还必须加密存储在 LDAP 服务器的密码。
- **hashEncoding=base64|hex**: 已加密密码的字符串格式, 或者为“base64”, 或者为“hex”。默认值为 base64。
- **hashCharset=String**: 用于转换明文密码到字节数组的编码(encoding)。默认值为平台默认编码。
- **allowEmptyPasswords**: 标志位, 表明是否允许传递长度为 0 的空密码给 LDAP 服务器。某些 LDAP 服务器将空密码作为匿名登录处理。如果为 false, 则拒绝空密码访问; 如果为 true, 则允许 LDAP 服务验证空密码。默认值为 true。

借助于登录模块配置选项提供的信息连接到 LDAP 服务器能够完成用户的认证过程。开发者使用由上述描述的 LDAP JNDI 属性组成的环境创建 InitialLdapContext 对象, 能够连接到 LDAP 服务器。其中, Context.SECURITY_PRINCIPAL 设置为用户的区别名, 通过合并 principalDNPrefix 和 principalDNSuffix 选项值能够获得该区别名。另外, 基于 userObjectCredential 选项, 将属性 Context.SECURITY_CREDENTIALS 设置为 String 形式的密码或者 Object 格式的凭证。

一旦借助于创建的 InitialLdapContext 实例成功完成用户认证, 则服务器会通过 roleAttributeName 和 uidAttributeName 选项值给出的搜索条件来搜索 rolesCtxDN 位置, 以查询用户角色。最后, 针对搜索结果集合的 role 属性, 服务器调用 toString 方法获得角色名。

Sun 遗留登录配置入口格式的实例如下:

```
testLdap {
    org.jboss.security.auth.spi.LdapLoginModule required
    java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory
```

```
java.naming.provider.url="ldap://ldaphost.jboss.org:1389/"
java.naming.security.authentication=simple
principalDNPrefix=uid=
uidAttributeID=userid
roleAttributeID=roleName
principalDNSuffix=,ou=People,o=jboss.org
rolesCtxDN=cn=JBossSX Tests,ou=Roles,o=jboss.org
};
```

对应的 XMLLoginConfig 格式为：

```
<policy>
  <application-policy name="testLdap">
    <authentication>
      <login-module code="org.jboss.security.auth.spi.LdapLoginModule"
        flag="required">
        <module-option name="java.naming.factory.initial">
          com.sun.jndi.ldap.LdapCtxFactory
        </module-option>
        <module-option name="java.naming.provider.url">
          ldap://ldaphost.jboss.org:1389/
        </module-option>
        <module-option name="java.naming.security.authentication">
          simple
        </module-option>
        <module-option name="principalDNPrefix">uid=</module-option>
        <module-option name="uidAttributeID">userid</module-option>
        <module-option name="roleAttributeID">roleName</module-option>
        <module-option name="principalDNSuffix">,ou=People,o=jboss.org
        </module-option>
        <module-option name="rolesCtxDN">cn=JBossSX Tests,ou=Roles,o=jboss.org
        </module-option>
      </login-module>
    </authentication>
  </application-policy>
</policy>
```

为帮助开发者理解 LdapLoginModule 的所有选项，请参考图 8-9 给出的 LDAP 服务器实例，该图对应于 testLdap 登录配置。

请注意比较图 8-9 给出的方案和上述列表给出的 testLdap 登录模块配置。java.naming.factory.initial、java.naming.factory.url 及 java.naming.security 选项表明，该配置使用了 Sun LDAP JNDI 供应商实现，LDAP 服务器位于主机 ldaphost.jboss.org（端口 1389），使用简单用户名和密码认证连接到 LDAP 服务器的客户。

当 LdapLoginModule 进行用户认证时，它连接到 java.naming.factory.url 指定的 LDAP 服务器。其中，使用了 principalDNPrefix 属性、传入的用户名及 principalDNSuffix 属性创建 java.naming.security.principal 属性值。对于 testLdap 配置而言，用户名“jduke”对应的

java.naming.security.principal 字符串为“uid=jduke,ou=People,o=jboss.org”，即对应于图 8-9 中右下角的 LDAP 上下文“Principal Context”。java.naming.security.credentials 属性值为传入的密码，对应于 Principal Context 中的 userPassword 属性。不同的 LDAP 服务器决定了受保护 LDAP 上下文存储认证凭证资料的具体方式。因此，不同 LDAP 服务器处理 java.naming.security.credentials 属性的有效性方式是不同的。

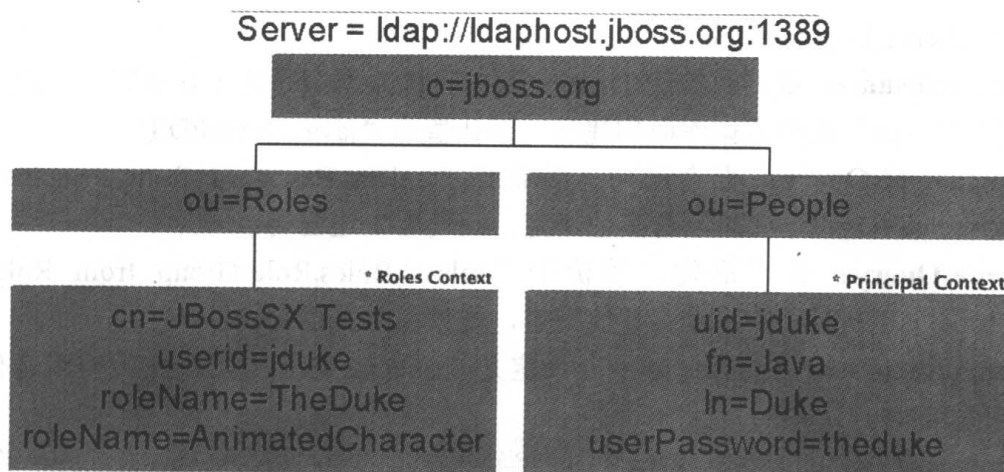


图 8-9 兼容 testLdap 实例配置的 LDAP 服务器配置

一旦客户成功通过认证，服务器通过搜索 LDAP 上下文便能够获得用户对应的角色。其中，该 LDAP 上下文的区别名由 rolesCtxDN 选项值给出。对于 testLdap 配置，上下文为“cn=JBossSX Tests,ou=Roles,o=jboss.org”，对应于图 8-9 中左下角的 LDAP 上下文“Roles Context”。该搜索将试图找到含有 uidAttributeID 选项指定属性，且属性值对应于传入登录模块的用户名的子上下文。对于任何匹配上下文，将获得由 roleAttributeID 选项指定属性名的所有取值。对于 testLdap 配置而言，含有角色的属性名称称之为 roleName。LdapLoginModule 会将返回的 roleName 值存储在 JAAS Subject 中，并将其作为角色组 Principal，用于基于角色的授权。对于图 8-9 给出的 LDAP 方案而言，分配给“jduke”用户的角色是“TheDuke”和“AnimatedCharacter”。

4. org.jboss.security.auth.spi.DatabaseServerLoginModule

DatabaseServerLoginModule 是基于 JDBC 实现并支持认证和角色映射的登录模块。如果将用户名、密码及角色信息都存储在 JDBC 使用的数据库中，则开发者可以使用 DatabaseServerLoginModule 登录模块。其中，该模块主要基于如下逻辑表：

Table Principals(PrincipalID text, Password text)

Table Roles(PrincipalID text, Role text, RoleGroup text)

Principals 表关联用户 PrincipalID 和有效密码，Roles 表关联用户 PrincipalID 和它的角色集合。用于用户许可的角色必须保存在 Roles 表 RoleGroup 列值的行中。这些表只是逻辑表，因此可以使用 SQL 查询获得用户认证信息。使用 DatabaseServerLoginModule 登录模块的惟一要求是返回的 java.sql.ResultSet 逻辑结构必须同 Principals 和 Roles 表相同。实际表名和列名没有关联基于列访问的结果集。比如，考虑具有如下两张表的数据库，即 Principals 和 Roles 表。如下的语句往 Principals 中添加 PrincipalID “java” 和 Password

“echoman”。另外，Roles 表为 PrincipalID (“java”) 插入了 RoleGroup (“Roles”) 中的角色 “Echo” 和 RoleGroup (“CallerPrincipal”) 中的角色 “caller_java”。

```
INSERT INTO Principals VALUES('java', 'echoman')
INSERT INTO Roles VALUES('java', 'Echo', 'Roles')
INSERT INTO Roles VALUES('java', 'caller_java', 'CallerPrincipal')
```

DatabaseServerLoginModule 登录配置支持的选项如下：

- **dsJndiName**: 数据源的 JNDI 名。其中，该数据源是基于含有逻辑 “Principals” 和 “Roles” 表的数据库而创建的。默认值为 “java:/DefaultDS”。
- **principalsQuery**: 查询语句，等价于 “select Password from Principals where PrincipalID=?”。如果没有其他指定，则使用上述语句。
- **rolesQuery**: 查询语句，等价于 “select Roles,RoleGroup from Roles where PrincipalID=?”。如果没有其他指定，则使用上述语句。
- **unauthenticatedIdentity=String**: 定义 Principal 名，即分配给没有包含任何认证信息的客户请求。
- **password-stacking=useFirstPass**: 当开发者设置了 password-stacking 选项时，DatabaseServerLoginModule 首先会在登录模块共享状态 Map 中分别寻找属性名 “javax.security.auth.login.name” 和 “javax.security.auth.login.password” 下的共享用户名和密码。如果找到，则将其作为 Principal 名和密码。如果没找到，则将 DatabaseServerLoginModule 设置的 Principal 名和密码存储在属性名 “javax.security.auth.login.name” 和 “javax.security.auth.login.password” 下。
- **hashAlgorithm=String**: java.security.MessageDigest 算法名，用于 HASH 加密。默认情况下服务器并没有给出该选项，因此必须给出该选项以生效 HASH 加密。当开发者指定 hashAlgorithm 时，首先会加密 CallbackHandler 返回的明文密码，然后传入 UsernamePasswordLoginModule.validatePassword 方法。其中，传入到上述方法的密码是以 inputPassword 参数形式传递的。最后，开发者还必须加密存储在数据库中的密码。
- **hashEncoding=base64|hex**: 已加密密码的字符串格式，或者为 “base64”，或者为 “hex”。默认值为 base64。
- **hashCharset=String**: 用于转换明文密码到字节数组的编码 (encoding)。默认值为平台默认编码。

请开发者参考某 DatabaseServerLoginModule 配置实例，其使用如下的自定义表：

```
CREATE TABLE Users(username VARCHAR(64) PRIMARY KEY, passwd VARCHAR(64))
CREATE TABLE UserRoles(username VARCHAR(64), userRoles VARCHAR(32))
```

对应 DatabaseServerLoginModule 配置的 Sun 遗留格式实例如下：

```
testDB {
    org.jboss.security.auth.spi.DatabaseServerLoginModule required
    dsJndiName="java:/MyDatabaseDS"
    principalsQuery="select passwd from Users username where username=?"
```

```

        rolesQuery="select userRoles, 'Roles' from UserRoles where username=?"
    };
};

```

对应的 XMLLoginConfig 配置如下:

```

<policy>
  <application-policy name="testDB">
    <authentication>
      <login-module
        code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
        flag="required">
        <module-option name="dsJndiName">java:/MyDatabaseDS
        </module-option>
        <module-option name="principalsQuery">select passwd from Users
        username where username=?</module-option>
        <module-option name="rolesQuery">select userRoles, 'Roles' from
        UserRoles where username=?</module-option>
      </login-module>
    </authentication>
  </application-policy>
</policy>

```

5. org.jboss.security.auth.spi.ProxyLoginModule

登录模块 ProxyLoginModule, 使用当前线程上下文类装载器装载委派 LoginModule。该登录模块的主要目的是解决当前 JAAS 1.0 类装载器的缺陷, 即它要求 LoginModule 处于系统类路径¹中。有些自定义 LoginModule 使用 JBoss 服务器 lib/ext 目录中的类, 因此如果 LoginModule 放置在系统类路径中, 这些类对于 LoginModule 不可见。为解决这个缺陷, 开发者可以使用 ProxyLoginModule 引导自定义 LoginModule。其中, ProxyLoginModule 存在称之为“moduleName”的配置选项。它指定待引导 LoginModule 实现的全限定类名。当然, 可以在 ProxyLoginModule 中指定其他配置选项, 并传入到被引导 LoginModule 中。

请开发者考虑如下实例, 即某自定义登录模块使用了 JBoss lib/ext 目录的类。自定义登录模块的类名是 com.biz.CustomServiceLoginModule。为引导这个自定义登录模块, 本文给出 Sun 遗留 ProxyLoginModule 配置入口的配置实例如下:

```

testProxy {
    org.jboss.security.auth.spi.ProxyLoginModule required
    moduleName=com.biz.CustomServiceLoginModule
    customOption1=value1
    customOption2=value2
    customOption3=value3;
};

```

¹ 由于 JBoss 3.x 提供的自身 JAAS 实现解决了这个问题, 从而不需要 ProxyLoginModule。另外, JDK 1.4 JAAS 实现也是如此。因此, 保留 ProxyLoginModule 的理由在于考虑到向后兼容性。

对应于 XMLLoginConfig 配置：

```
<policy>
  <application-policy name="testProxy">
    <authentication>
      <login-module code="org.jboss.security.auth.spi.ProxyLoginModule"
        flag="required">
        <module-option name="moduleName">com.biz.CustomServiceLoginModule
        </module-option>
        <module-option name="customOption1">value1</module-option>
        <module-option name="customOption2">value2</module-option>
        <module-option name="customOption3">value3</module-option>
      </login-module>
    </authentication>
  </application-policy>
</policy>
```

6. org.jboss.security.auth.spi.RunAsLoginModule

RunAsLoginModule 是 JBoss 3.0.3 引入的辅助登录模块。它为认证的 login 阶段压入 run-as 角色，commit 或 abort 阶段弹出 run-as 角色。该模块的主要目的是为其他登录模块提供访问受保护资源以完成认证过程的角色，比如某登录模块需要访问受保护的 EJB。其中，开发者必须将该登录模块配置在需要建立 run-as 角色的登录模块前。

该登录模块的惟一配置选项如下：

- **roleName:** 用于 login 阶段的 run-as 角色名。默认值为 “nobody”。

7. org.jboss.security.ClientLoginModule

登录模块 ClientLoginModule 供 JBoss 客户建立调用者身份和凭证使用。其中，它简单地将 org.jboss.security.SecurityAssociation.principal 和 org.jboss.security.SecurityAssociation.credential 分别设置为 CallbackHandler 填充的 NameCallback 和 PasswordCallback 值。这也是客户用于建立当前线程调用者的惟一支持机制。对于那些充当 JBoss EJB 客户的单独客户应用和服务器环境，如果安全环境没有配置为透明使用 JBossSX，则需要使用 ClientLoginModule。当然，开发者也可以直接设置 org.jboss.security.SecurityAssociation 信息，但这是内部 API，不需要关注它是否发生变化。

请开发者注意，该模块并不完成任何认证工作，它只是拷贝登录信息给 JBoss 服务器的 EJB 调用层，供服务器后续认证使用。如果开发者打算实现用户的客户端认证，则除了配置 ClientLoginModule 模块外，还需要其他的登录模块。

ClientLoginModule 登录模块配置支持的选项如下：

- **multi-threaded=true|false:** 当设置为 true 时，每个登录线程都有自己的 Principal 和凭证存储源。对于单独线程存在多个活动用户身份的客户环境而言，该属性特别有用。当设置为 true 时，各个线程必须完成各自的登录操作。当设置为 false 时，登录身份和凭证是全局变量，即 JVM 中的所有线程共享它们。默认值为 false。
- **password-stacking=useFirstPass:** 当开发者设置了 password-stacking 选项时，

ClientLoginModule 首先会在登录模块共享状态 Map 中分别寻找属性名“javax.security.auth.login.name”和“javax.security.auth.login.password”下的共享用户名和密码。如果找到，则将其作为 Principal 名和密码。如果没找到，则将 ClientLoginModule 设置的 Principal 名和密码存储在属性名“javax.security.auth.login.name”和“javax.security.auth.login.password”下。如果开发者使用其他的登录模块，比如 LdapLoginModule，完成客户的客户端认证，则应使用该选项。

比如，JBoss 发布版的 client/auth.conf 文件中包含了 ClientLoginModule 的默认登录配置入口。其具体内容如下：

```
other {  
    // Put your login modules that work without JBoss here  
  
    // jBoss LoginModule  
    org.jboss.security.ClientLoginModule required;  
  
    // Put your login modules that need JBoss here  
};
```

8.4.7 开发自定义登录模块

如果 JBossSX 框架绑定的登录模块不能够满足目标安全性环境，则开发者需要开发自定义的登录模块实现。

前面在论述 JaasSecurityManager 架构时提到，JaasSecurityManager 包含了使用 Subject Principal 集合的特定模式。开发者需要掌握 JAAS Subject 类的信息存储特征及这些特性的使用方式，才能够同 JaasSecurityManager 协同工作的登录模块。本节内容将研究上述需求，并介绍两个抽象 LoginModule 基本实现，帮助开发者实现自定义登录模块。

开发者可以使用如下 6 种方式获得 Subject 关联的安全性信息：

```
java.util.Set getPrincipals()  
java.util.Set getPrincipals(java.lang.Class c)  
java.util.Set getPrivateCredentials()  
java.util.Set getPrivateCredentials(java.lang.Class c)  
java.util.Set getPublicCredentials()  
java.util.Set getPublicCredentials(java.lang.Class c)
```

对于 Subject 身份和角色，JBossSX 选择了两种最符合的方法获得 Principal 集合：getPrincipals()和 getPrincipals(java.lang.Class)。具体使用模式如下：

- 用户身份（用户名、社会保险号、雇员号等）被存储为 Subject Principal 集合中的 java.security.Principal 对象。其中，代表用户身份的 Principal 实现必须能够实现 Principal 名的比较和等于操作。比如，某合适的实现为 org.jboss.security.SimplePrincipal 类。如果需要，开发者也可以添加其他 Principal 实例给 Subject Principal 集合。

- 分配的用户角色也存储在 Principal 集合中，但是使用了 java.security.acl.Group 实例将它们分组为角色集合。Group 接口定义了 Principal 和（或）Group，它是 java.security.Principal 的子接口。可以为 Subject 分配任何数量的角色集合。当前，JBossSX 框架使用两种众所周知的角色集合：Roles 和 CallerPrincipal。其中，“Roles” Group 集合了应用域中已通过认证 Subject 的所有 Principal。这部分角色集合供方法使用，比如方法 EJBContext.isCallerInRole(String)，EJB 从而能够判断当前调用者是否属于指定应用域角色。同时，完成方法许可检查的安全性拦截器逻辑也要使用这部分角色集合。“CallerPrincipal” Group 由应用域中分配给用户的单个 Principal 组成。EJBContext.getCallerPrincipal()方法使用“CallerPrincipal”以实现应用域映射，即将操作环境身份映射为适合于应用的用户身份。如果 Subject 不存在“CallerPrincipal” Group，则应用身份和操作环境身份相同。

1. 支持 Subject 使用模式

为简化 Subject 使用模式的正确实现（前面提到过），JBossSX 包括了两个抽象登录模块，即提供了模板模式以处理认证 Subject，从而加强 Subject 的正确使用。在这两个模块中，org.jboss.security.auth.spi.AbstractServerLoginModule 类最通用化。它提供了接口 javax.security.auth.spi.LoginModule 的具体实现，并提供了与操作环境安全性基础框架相关的主要任务的抽象方法。该类的重要细节在下面的类片段中以高亮显示。其中的具体细节请参考 javadoc 文档。

```
package org.jboss.security.auth.spi;

/** This class implements the common functionality required for a
JAAS server-side LoginModule and implements the JBossSX standard
Subject usage pattern of storing identities and roles. Subclass
this module to create your own custom LoginModule and override the
login(), getRoleSets(), and getIdentity() methods.
*/

public abstract class AbstractServerLoginModule
implements javax.security.auth.spi.LoginModule
{
    protected Subject subject;
    protected CallbackHandler callbackHandler;
    protected Map sharedState;
    protected Map options;
    protected Logger log;
    /** Flag indicating if the shared credential should be used */
    protected boolean useFirstPass;
    /** Flag indicating if the login phase succeeded. Subclasses that override
the login method must set this to true on successful completion of login
*/
    protected boolean loginOk;

    ...

    /**
```

```

* Initialize the login module. This stores the subject, callbackHandler
* and sharedState and options for the login session. Subclasses should
  override
* if they need to process their own options. A call to
  super.initialize(...)
* must be made in the case of an override.
* <p>
* The options are checked for the <em>password-stacking</em> parameter.
* If this is set to "useFirstPass", the login identity will be taken from
  the
* <code>javax.security.auth.login.name</code> value of the sharedState
  map,
* and the proof of identity from the
* <code>javax.security.auth.login.password</code> value of the
  sharedState map.
*
* @param subject the Subject to update after a successful login.
* @param callbackHandler the CallbackHandler that will be used to obtain
  the
* the user identity and credentials.
* @param sharedState a Map shared between all configured login module
  instances
* @param options the parameters passed to the login module.
*/

```

```

public void initialize(Subject subject,
  CallbackHandler callbackHandler,
  Map sharedState,
  Map options)
{
  ...
}

```

```

/** Looks for javax.security.auth.login.name and
  javax.security.auth.login.password
  values in the sharedState map if the useFirstPass option was true and
  returns
  true if they exist. If they do not or are null this method returns false.

```

```

Note that subclasses that override the login method must set the loginOk
ivar to true if the login succeeds in order for the commit phase to
populate the Subject. This implementation sets loginOk to true if the
login() method returns true, otherwise, it sets loginOk to false.

```

```

*/
public boolean login() throws LoginException
{
  ...
}

```

```
}

/** Overridden by subclasses to return the Principal that
corresponds to the user primary identity.
*/
abstract protected Principal getIdentity();

/** Overridden by subclasses to return the Groups that
correspond to the role sets assigned to the user. Subclasses
should create at least a Group named "Roles" that contains
the roles assigned to the user.
A second common group is "CallerPrincipal," which provides
the application identity of the user rather than the security
domain identity.
@return Group[] containing the sets of roles
*/
abstract protected Group[] getRoleSets() throws LoginException;
}
```

JBoss 3.0.3 的一个重要变化是新增了 loginOk 实例变量。如果 login 方法成功执行, 则覆盖了 login 方法的子类将设置 loginOk 为 true, 否则将设置为 false。如果没有正确地设置该变量, 将导致 commit 方法无法判断是否应该更新 Subject。这种对 login 阶段的跟踪能够使用控制标志将登录模块连在一起。

org.jboss.security.auth.spi.UsernamePasswordLoginModule 类是第二个适合于自定义登录模块的抽象登录模块基本实现。它进一步简化了自定义登录模块实现, 即强行将基于字符串形式的用户名作为用户身份、char[]形式的密码作为认证凭证。它也支持匿名用户 (null 用户名和密码) 到没有角色的 Principal 的映射。该类的重要细节在下面的类片段中以高亮显示。其中的具体细节请参考 javadoc 文档。

```
package org.jboss.security.auth.spi;

/** An abstract subclass of AbstractServerLoginModule that imposes
a an identity == String username, credentials == String password
view on the login process. Subclasses override the
getUsersPassword() and getUsersRoles() methods to return the
expected password and roles for the user.
*/
public abstract class UsernamePasswordLoginModule
extends AbstractServerLoginModule
{
    /** The login identity */
    private Principal identity;
    /** The proof of login identity */
    private char[] credential;
    /** The principal to use when a null username and password
are seen */
    private Principal unauthenticatedIdentity;
```

```
/** The message digest algorithm used to hash passwords. If null then
plain passwords will be used. */
private String hashAlgorithm = null;
/** The name of the charset/encoding to use when converting the password
String to a byte array. Default is the platform's default encoding.
*/
private String hashCharset = null;
/** The string encoding format to use. Defaults to base64. */
private String hashEncoding = null;

...

/** Override the superclass method to look for an
unauthenticatedIdentity property. This method first invokes
the super version.
@param options,
@option unauthenticatedIdentity: the name of the principal
to assign and authenticate when a null username and password
are seen.
*/
public void initialize(Subject subject,
CallbackHandler callbackHandler,
Map sharedState,
Map options)
{
    super.initialize(subject, callbackHandler, sharedState,
options);
    // Check for unauthenticatedIdentity option.
    Object option = options.get("unauthenticatedIdentity");
    String name = (String) option;
    if( name != null )
        unauthenticatedIdentity = new SimplePrincipal(name);
}

...

/** A hook that allows subclasses to change the validation of
the input password against the expected password. This version
checks that neither inputPassword or expectedPassword are null
and that inputPassword.equals(expectedPassword) is true;
@return true if the inputPassword is valid, false otherwise.
*/
protected boolean validatePassword(String inputPassword,
String expectedPassword)
{
    if( inputPassword == null || expectedPassword == null )
```



```
        return false;
        return inputPassword.equals(expectedPassword);
    }

    /** Get the expected password for the current username
     *  available via the getUsername() method. This is called from
     *  within the login() method after the CallbackHandler has
     *  returned the username and candidate password.
     *  @return the valid password String
     */
    abstract protected String getUsersPassword()
        throws LoginException;
}
```

具体选择 `AbstractServerLoginModule` 还是 `UsernamePasswordLoginModule`，简单地取决于基于 `String` 的用户名和凭证是否适用于待开发登录模块所要求的认证技术。如果基于 `String` 的语义有效，则实现 `UsernamePasswordLoginModule` 的子类，否则实现 `AbstractServerLoginModule` 的子类。

根据选择登录模块基类的不同，下面给出了不同的开发自定义登录模块的具体步骤。当开发自定义登录模块以集成到目标安全性基础框架中时，应该实现 `AbstractServerLoginModule` 或 `UsernamePasswordLoginModule`，才能够保证登录模块提供了 JBossSX 安全性管理器期望的认证 `Principal` 信息格式。

当实现 `AbstractServerLoginModule` 子类时，开发者需要重载如下方法：

- `void initialize(Subject, CallbackHandler, Map, Map)`: 如果需要分析自定义选项。
- `boolean login()`: 完成认证行为。如果 `login` 方法执行成功，则务必将 `loginOk` 实例变量设为 `true`，否则设为 `false`。
- `Principal getIdentity()`: 返回 `login()` 步骤认证用户的 `Principal` 对象。
- `Group[] getRoleSets()`: 至少要返回一个称为“Roles”的 `Group`，其中该 `Group` 含有 `login()` 期间分配给已认证 `Principal` 的角色。另一个“CallerPrincipal” `Group` 提供了用户的应用身份，而不是安全域身份。

当实现 `UsernamePasswordLoginModule` 子类时，需要重载如下方法：

- `void initialize(Subject, CallbackHandler, Map, Map)`: 如果需要分析自定义选项。
- `Group[] getRoleSets()`: 至少要返回一个称为“Roles”的 `Group`，其中该 `Group` 含有 `login()` 期间分配给已认证 `Principal` 的角色。另一个“CallerPrincipal” `Group` 提供了用户的应用身份，而不是安全域身份。
- `String getUsersPassword()`: 借助于 `getUsername()` 方法返回当前用户名的期望密码。在 `CallbackHandler` 返回用户名和候选密码后，`login()` 会调用 `getUsersPassword()` 方法。

2. 自定义 LoginModule 实例

接下来将开发自定义登录模块的实例。它将扩展 `UsernamePasswordLoginModule`，并从 JNDI 查找中获得用户的密码和角色名。思路如下：使用“password/<username>”形式的名字，其中<username>为当前认证用户，查找 JNDI 上下文，将返回用户密码。类似地，

“roles/<username>”形式的查找返回请求用户的角色。

源代码位于 src/main/org/jboss/chap8/ex2 目录下。列表 8-10 给出了其源代码，即 JndiUserAndPass 自定义登录模块。需要注意的是，由于它扩展了 JBoss UsernamePassword LoginModule，所有 JndiUserAndPass 是从 JNDI 存储源中获得用户密码和角色。JndiUserAndPass 本身没有提供 JAAS LoginModule 操作。

列表 8-10 JndiUserAndPass 自定义登录模块

```
package org.jboss.chap8.ex2;

import java.security.acl.Group;
import java.util.Map;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginException;

import org.jboss.security.SimpleGroup;
import org.jboss.security.SimplePrincipal;
import org.jboss.security.auth.spi.UsernamePasswordLoginModule;

/** An example custom login module that obtains passwords and roles for a
    user
    from a JNDI lookup.
    @author Scott.Stark@jboss.org
    @version $Revision$
    */
public class JndiUserAndPass extends UsernamePasswordLoginModule
{
    /** The JNDI name to the context that handles the password/<username>
        lookup */
    private String userPathPrefix;
    /** The JNDI name to the context that handles the roles/<username> lookup
        */
    private String rolesPathPrefix;

    /** Override to obtain the userPathPrefix and rolesPathPrefix options.
        */
    public void initialize(Subject subject, CallbackHandler callbackHandler,
        Map sharedState, Map options)
    {
        super.initialize(subject, callbackHandler, sharedState, options);
        userPathPrefix = (String) options.get("userPathPrefix");
        rolesPathPrefix = (String) options.get("rolesPathPrefix");
    }
}
```

```
/** Get the roles the current user belongs to by querying the
rolesPathPrefix + '/' + super.getUsername() JNDI location.
*/
protected Group[] getRoleSets() throws LoginException
{
    try
    {
        InitialContext ctx = new InitialContext();
        String rolesPath = rolesPathPrefix + '/' + super.getUsername();
        String[] roles = (String[]) ctx.lookup(rolesPath);
        Group[] groups = {new SimpleGroup("Roles")};
        log.info("Getting roles for user="+super.getUsername());
        for(int r = 0; r < roles.length; r++)
        {
            SimplePrincipal role = new SimplePrincipal(roles[r]);
            log.info("Found role="+roles[r]);
            groups[0].addMember(role);
        }
        return groups;
    }
    catch(NamingException e)
    {
        log.error("Failed to obtain groups for user="+super.getUsername(),e);
        throw new LoginException(e.toString(true));
    }
}

/** Get the password of the current user by querying the
userPathPrefix + '/' + super.getUsername() JNDI location.
*/
protected String getUsersPassword() throws LoginException
{
    try
    {
        InitialContext ctx = new InitialContext();
        String userPath = userPathPrefix + '/' + super.getUsername();
        log.info("Getting password for user="+super.getUsername());
        String passwd = (String) ctx.lookup(userPath);
        log.info("Found password="+passwd);
        return passwd;
    }
    catch(NamingException e)
    {
        log.error("Failed to obtain password for user="+super.getUsername(),e);
    }
}
```

```

        throw new LoginException(e.toString(true));
    }
}
}

```

JNDI 存储的具体细节请参考 `org.jboss.chap8.ex2.service.JndiStore` MBean。该服务绑定 `ObjectFactory`，即返回 `javax.naming.Context` 代理给 JNDI。该代理基于查找名前缀“password”和“roles”处理查找操作。当查找名以“password”开始，则请求用户密码。当查找名以“roles”开始，则请求用户角色。该实例不管用户名是什么，总是返回密码“theduke”和角色名数组{“TheDuke”，“Echo”}。开发者可以尝试其他实现。

实例代码包含了简单会话 Bean，以测试该自定义 `LoginModule`。从光盘目录执行如下命令行能够编译、部署及运行实例。一定要让 JBoss 服务器运行。列表 8-11 给出了客户的主要输出行，而列表 8-12 给出了服务器端的输出行。

列表 8-11 chap8-ex2 保护客户访问输出

```

[nr@toki examples]$ ant -Dchap=chap8 -Dex=2 run-example
Buildfile: build.xml
...
run-example2:
    [copy] Copying 1 file to /tmp/jboss-3.2.3/server/default/deploy
    [echo] Waiting for 5 seconds for deploy...
    [java] [INFO,ExClient] Login with username=jduke, password=theduke
    [java] [INFO,ExClient] Looking up EchoBean2
    [java] [INFO,ExClient] Created Echo
    [java] [INFO,ExClient] Echo.echo('Hello') = Hello

```

BUILD SUCCESSFUL

Total time: 12 seconds

列表 8-12 chap8-ex2 服务器端 JndiUserAndPass 行为

```

01:34:11,118 INFO [MainDeployer] Starting deployment of package: file:/private/tmp/jboss-3.2.3/
server/default/deploy/chap8-ex2.jar
01:34:11,312 INFO [EJBDeployer] nested deployment: file:/private/tmp/jboss-3.2.3/
server/default/tmp/deploy/tmp36177chap8-ex2.jar-contents/chap8-ex2.sar
01:34:12,831 INFO [EjbModule] Deploying EchoBean2
01:34:13,084 INFO [JaasSecurityManagerService] Created securityMgr=org.jboss.security.plugins.
JaasSecurityManager@1df832
01:34:13,091 INFO [JaasSecurityManagerService] setCachePolicy, c=org.jboss.util.TimedCache Policy@91b2c8
01:34:13,094 INFO [JaasSecurityManagerService] Added chap8-ex2, org.jboss.security.plugins.
SecurityDomainContext@7095e5 to map
01:34:13,849 INFO [JndiStore] Start, bound security/store
01:34:13,854 INFO [SecurityConfig] Using JAAS AuthConfig: jar:file:/private/tmp/jboss-3.2.3/
server/default/tmp/deploy/tmp36177chap8-ex2.jar-contents/chap8-ex2.sar!/META-INF/login-
config.xml

```



```
01:34:14,174 INFO [SecurityConfig] Started jboss.docs.chap8:service=LoginConfig-EX2
01:34:14,679 INFO [StatelessSessionInstancePool] Started jboss.j2ee:jndiName=EchoBean2,
plugin=pool,service=EJB
01:34:14,683 INFO [StatelessSessionContainer] Started jboss.j2ee:jndiName=EchoBean2, service=EJB
01:34:14,687 INFO [EjbModule] Started jboss.j2ee:module=chap8-ex2.jar,service=EjbModule
01:34:14,690 INFO [EJBDeployer] Deployed: file:/private/tmp/jboss-3.2.3/server/
default/deploy/chap8-ex2.jar
01:34:15,043 INFO [MainDeployer] Deployed package: file:/private/tmp/jboss-3.2.3/server/
default/deploy/chap8-ex2.jar
01:34:19,685 INFO [JndiUserAndPass] Getting password for user=jduke
01:34:19,859 INFO [JndiStore] lookup, name=password/jduke
01:34:19,862 INFO [JndiUserAndPass] Found password=theduke
01:34:19,940 INFO [JndiStore] lookup, name=roles/jduke
01:34:19,946 INFO [JndiUserAndPass] Getting roles for user=jduke
01:34:19,947 INFO [JndiUserAndPass] Found role=TheDuke
01:34:19,947 INFO [JndiUserAndPass] Found role=Echo
```

安全性域实例配置的登录配置使用了 JndiUserAndPass 自定义登录模块作为服务器端认证。ejb-jar META-INF/jboss.xml 描述符设置安全性域，sar META-INF/login-config.xml 描述符定义登录模块配置。列表 8-13 给出了这些描述符的内容。

列表 8-13 chap8-ex2 安全性域和登录模块配置

chap8-ex2 jboss.xml 描述符安全性域设置

```
<?xml version="1.0"?>
<jboss>
  <security-domain>java:/jaas/chap8-ex2</security-domain>
</jboss>
```

用于 chap8-ex2 应用的 login-config.xml 配置片段

```
<application-policy name = "chap8-ex2">
  <authentication>
    <login-module code = "org.jboss.chap8.ex2.JndiUserAndPass"
      flag = "required">
      <module-option name = "userPathPrefix">/security/store/password
    </module-option>
      <module-option name = "rolesPathPrefix">/security/store/roles
    </module-option>
    </login-module>
  </authentication>
</application-policy>
```

8.5 安全远程密码协议

安全远程密码 (Secure Remote Password, SRP) 协议实现了公钥交换握手。其中, 公钥交换握手内容由 Internet Standards Working Group 请求评议 2945 (RFC2945) 阐述。摘入 RFC2945 如下:

本文档描述安全性极高的认证机制, 即著名的安全远程密码 (Secure Remote Password, SRP) 协议。本机制适合于使用用户提供的密码协商保护连接, 并消除了传统使用可重用密码而带来的安全性问题。本系统在认证过程中也会完成安全密钥交换, 其中的会话过程允许使用安全性层 (私和, 或集成性保护)。不需要可信任密钥服务器和证书基础框架, 客户也不用存储或管理任何很大的密钥。因此, SRP 相比现有提问-响应技术而言, 它提供了安全性和部署优势。从而, 当需要使用安全密码认证时, 它能够替代现有的其他技术。



完整的 RFC2945 规范可以通过 <http://www.rfc-editor.org/rfc.html> 获得。
SRP 算法及其历史信息可以通过 <http://www-cs-students.stanford.edu/~tjw/srp/> 获得。

SRP 在概念和安全性上类似于其他公钥交换算法, 比如 Diffie-Hellman 和 RSA。SRP 基于简单的字符串密码, 不需要在服务器上存在明文密码。其他基于公钥的算法同 SRP 相比, 它们还需要客户证书和相应的认证管理基础框架。

类似于 Diffie-Hellman 和 RSA 的算法, 称为公钥交换算法。公钥算法一共存在两个密钥, 一个为公钥, 任何人可见; 另一个为私钥, 只有持有人知道它。当某人发送加密信息给密钥持有人时, 他会使用公钥加密这些待发送信息。只有密钥持有人才能够使用私钥解密信息。同更传统的基于密码加密方案相比, 它们要求发送者和接收者都知道共享密码。公钥算法消除了共享密码的需求。更多公钥算法和其他加密算法, 请参考由 Bruce Schneier 著的《Applied Cryptography (第二版)》图书 (ISBN 0-471-11709-9)。

JBossSX 框架包括的 SRP 实现由下列元素组成:

- SRP 握手协议实现, 独立于任何特定的客户/服务器协议。
- 握手协议的 RMI 实现, 并将它作为默认的客户/服务器 SRP 实现。
- 客户端 JAAS LoginModule 实现使用了该 RMI 实现, 以安全的方式来认证客户。
- 管理 RMI 服务器实现的 JMX MBean。该 MBean 允许 RMI 服务器实现插入到 JMX 框架中, 并实现认证信息存储源的外部配置。它还创建了认证缓存, 并将其绑定到 JBoss 服务器 JNDI 命名空间中。
- 服务器端 JAAS LoginModule 实现使用了由 SRP JMX MBean 管理的认证缓存。

SRP 客户/服务器框架 JBossSX 实现中重要组件的流程图, 如图 8-10 所示。

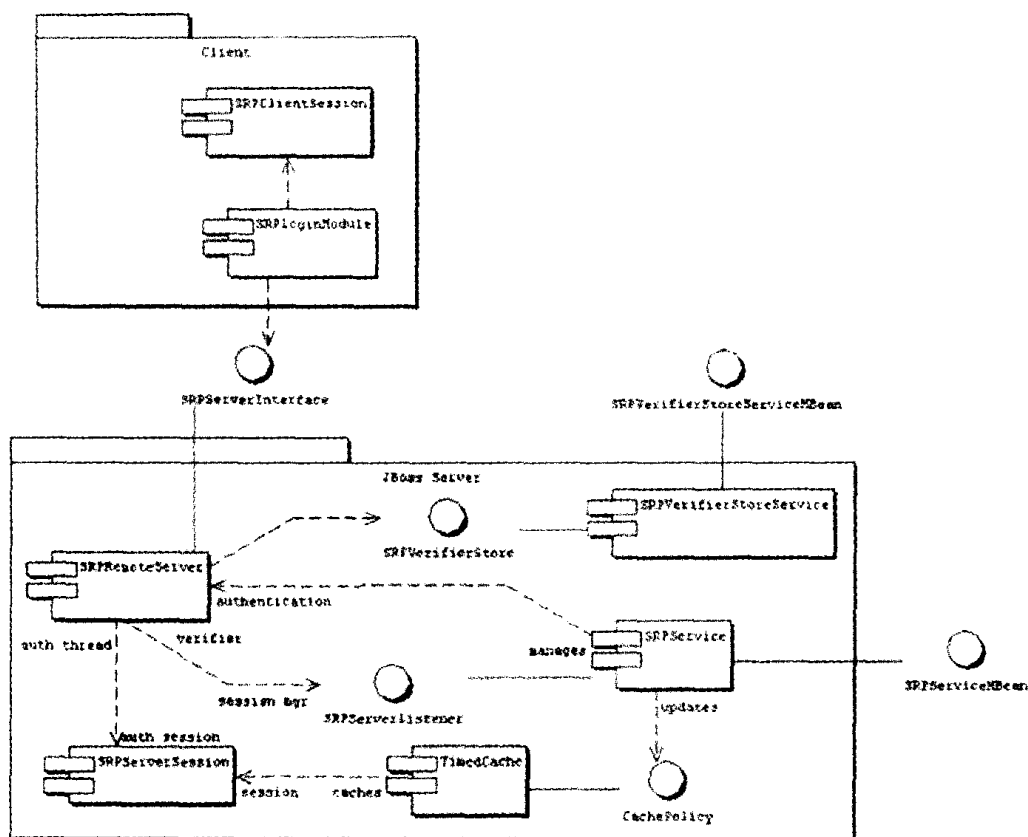


图 8-10 SRP 客户/服务器框架的 JBossSX 组件

在 SRP 客户端，SRP 通过自定义 JAAS LoginModule 实现暴露给客户，其中借助于 `org.jboss.security.srp.SRPServerInterface` 代理实现与认证服务器的通信。客户通过创建包含 `org.jboss.security.srp.jaas.SRPLoginModule` 的登录配置入口，即使用 SRP 实现认证。该模块支持的配置选项如下：

- **principalClassName:** 不再支持该选项，org.jboss.security.srp.jaas.SRPPrincipal 是该属性的固定值。
- **srpServerJndiName:** SRPServerInterface 对象的 JNDI 名，用于实现与 SRP 认证服务器的通信。如果同时指定 srpServerJndiName 和 srpServerRmiUrl 选项，则会先尝试 srpServerJndiName。
- **srpServerRmiUrl:** 用于定位 SRPServerInterface 代理，即实现与 SRP 认证服务器的通信的 RMI 协议 URL 字符串。
- **externalRandomA:** true/false 标志位，表明客户公钥的随机组件是否应该来自于用户回调。比如，可以使用它输入来自硬件令牌的、安全性极高的随机数。
- **hasAuxChallenge:** true/false 标志位，表明是否发送字符串到服务器作为额外请求以等待服务器验证。如果客户会话支持加密密码，则将使用会话私钥和发送为 javax.crypto.SealedObject 的请求对象创建临时密码。
- **multipleSessions:** true/false 标志位，表明特定客户是否能够同时激活多个 SRP 登录会话。

不匹配上述选项的其他传入值将被当做 JNDI 属性使用，以传递这些环境属性到

InitialContext 的构建器。如果默认 InitialContext 不能够获得 SRP 服务器接口，则这是一种很有用的机制。

SRPLoginModule 需要同标准 ClientLoginModule 一起配置，使得 SRP 认证凭证能够用于验证对 J2EE 安全性组件的访问。比如，如下给出了登录配置入口实例：

```
srp {  
    org.jboss.security.srp.jaas.SRPLoginModule required  
        srpServerJndiName="SRPServerInterface"  
    ;  
  
    org.jboss.security.ClientLoginModule required  
        password-stacking="useFirstPass"  
    ;  
};
```

在 JBoss 服务器端，存在两个 MBean 管理 SRP 服务器的组成对象。主要服务是 org.jboss.security.srp.SRPService MBean，它负责暴露 SRPServerInterface 的 RMI 可访问版本和更新 SRP 认证会话缓存。SRPService MBean 的可配置属性如下：

- **JndiName:** SRPServerInterface 代理绑定的 JNDI 名。它是 SRPService 将序列化动态代理绑定到 SRPServerInterface 的目标位置。默认值为“srp/SRPServerInterface”。
- **VerifierSourceJndiName:** SRPVerifiedSource 实现的 JNDI 名，供 SRPService 使用。默认值为“srp/DefaultVerifierSource”。
- **AuthenticationCachedJndiName:** 认证 org.jboss.util.CachePolicy 实现的 JNDI 名，用于绑定缓存认证信息。SRP 会话缓存也是通过该绑定获得的。默认值为“srp/AuthenticationCache”。
- **ServerPort:** SRPRemoteServerInterface 的 RMI 端口。默认值为“10099”。
- **ClientSocketFactory:** 可选自定义 java.rmi.server.RMIClientSocketFactory 实现类名，供导出 SRPServerInterface 期间使用。默认值为“RMIClientSocketFactory”。
- **ServerSocketFactory:** 可选自定义 java.rmi.server.RMIServerSocketFactory 实现类名，供导出 SRPServerInterface 期间使用。默认值为“RMIServerSocketFactory”。
- **AuthenticationCacheTimeout:** 指定定时缓存策略的超时时间（单位：秒）。默认值为“1800 秒（30 分钟）”。
- **AuthenticationCacheResolution:** 指定默认定时缓存策略精度（单位：秒）。它用于控制检查超时的时间间隔。默认值为“60 秒（1 分钟）”。
- **RequireAuxChallenge:** 客户端是否必须提供额外请求，以作为认证的部分内容。因此，该选项能够控制客户使用的 SRPLoginModule 配置是否必须生效 hasAuxChallenge 选项。
- **OverwriteSessions:** 标志位，表明成功完成用户认证的现有会话是否应该覆盖当前会话。当客户没有生效“每用户多会话模式”时，它能够控制服务器 SRP 会话缓存的行为。默认值为“false”，其含义为，如果用户成功通过第二次认证，返回的 SRP 会话不会覆盖上次 SRP 会话状态。

VerifierSourceJndiName 属性能够输入设置信息。它指定和访问 SRP 密码信息存储源实现的位置，通过 JNDI 能够操作它。实例 MBean 服务，即 org.jboss.security.srp.SRPVerifierStore Service，绑定了 SRPVerifierStore 接口实现，该实现使用序列化对象文件作为持久化存储源。尽管该服务不能够用于产品环境，但还是能够测试 SRP 协议，并为 SRPVerifierStore 服务提供示范需求。SRPVerifierStoreService MBean 包括的可配置属性如下：

- **JndiName:** JndiName 属性指，SRPVerifierStore 实现绑定的 JNDI 名。默认值为“srp/DefaultVerifierSource”。
- **StoreFile:** StoreFile 属性用于指定如下文件位置，即用户密码验证器使用的序列化对象存储文件。它可能是 URL 或类路径中的资源名。默认值为“SRPVerifierStore.ser”。

SRPVerifierStoreService MBean 也为新增和删除用户提供了 addUser 和 delUser 操作。方法定义和命名如下：

```
public void addUser(String username, String password) throws IOException;  
public void delUser(String username) throws IOException;
```

这些服务的实例配置能够在“8.5 安全远程密码协议”内容中找到。

8.5.1 为 SRP 提供密码信息

SRPVerifierStore 接口的默认实现可能不适合于产品安全性环境，因为它将所有密码 HASH 信息都存储在序列化对象文件中。开发者需要提供 MBean 服务，即实现 SRPVerifierStore 接口，以集成到现有的安全性信息存储源中。其中，SRPVerifierStore 接口定义如列表 8-14 所示。

列表 8-14 SRPVerifierStore 接口

```
package org.jboss.security.srp;  
  
import java.io.IOException;  
import java.io.Serializable;  
import java.security.KeyException;  
  
public interface SRPVerifierStore  
{  
    public static class VerifierInfo implements Serializable  
    {  
        /** The username the information applies to. Perhaps redundant but it  
        makes the object self contained.  
        */  
        public String username;  
        /** The SRP password verifier hash */  
        public byte[] verifier;  
        /** The random password salt originally used to verify the password */  
        public byte[] salt;
```

```

    /** The SRP algorithm primitive generator */
    public byte[] g;
    /** The algorithm safe-prime modulus */
    public byte[] N;
}
/** Get the indicated user's password verifier information.
 */
public VerifierInfo getUserVerifier(String username)
    throws KeyException, IOException;
/** Set the indicated users' password verifier information. This is equivalent
to changing a user's password and should generally invalidate any
existing SRP sessions and caches.
 */
public void setUserVerifier(String username, VerifierInfo info)
    throws IOException;

/** Verify an optional auxillary challenge sent from the client to the server.
The auxChallenge object will have been decrypted if it was sent encrypted
from the client. An example of a auxillary challenge would be the validation of
a hardware token (SafeWord, SecureID, iButton) that the server validates to
further strengthen the SRP password exchange.
 */
public void verifyUserChallenge(String username, Object auxChallenge)
    throws SecurityException;
}

```

SRPVerifierStore 接口实现的主要功能是为特定用户名提供 SRPVerifierStore.VerifierInfo 对象访问。为获得 SRP 算法所需参数，在用户 SRP 会话开始时，SRPService 会调用 getUserVerified(String) 方法。VerifierInfo 对象的元素如下：

- **username:** 用户用于登录的名字或 ID。
- **verifier:** 这是用户输入的单工 HASH 密码或 PIN，以证明其身份。其中，org.jboss.security.Util 类提供的 calculateVerifier 方法实现了 HASH 密码算法，RFC2945 定义了其输出密码，即 $H(\text{salt} \parallel H(\text{username} \parallel ':' \parallel \text{password}))$ 。其中，H 指 SHA 的安全散列函数。它使用 UTF-8 编码将用户名从字符串格式转换为 byte[] 格式。
- **salt:** 随机数。如果数据库受到侵犯，则它将用于增加对验证器密码数据库的字典强力攻击的难度。当散列了用户的现有明文密码时，需要借助于安全性极高的随机数算法生成该值。
- **g:** SRP 算法原始生成器。通常，它只是固定的参数，而不是针对每用户提供不同的参数。借助于 org.jboss.security.srp.SRPConf 实用类的 getDefaultParams().g() 方法能够为 g 提供若干设置。
- **N:** SRP 算法主安全系数。通常，它只是固定的参数，而不是针对每用户提供不同的参数。借助于 org.jboss.security.srp.SRPConf 实用类的 getDefaultParams().N()

方法能够为 N 提供若干设置。

因此，集成现有密码存储源的第一步是创建密码信息的散列版本。如果密码已经是散列形式存储，并且不能够取消，则需要将其升级到基于各个用户的散列形式。请注意，当前 SRPService 并没有使用 setUserVerified(String,VerifierInfo)方法，而且有可能将会实现为无操作内容（noop）方法，甚至抛出异常以提示该存储源是只读的。

第二步，创建自定义 SRPVerifierStore 接口实现，从而能够从第一步创建的存储源获得 VerifierInfo。如果客户 SRPLoginModule 配置指定了 hasAuxChallenge 选项，则才会调用 SRPVerifierStore 接口的 verifyUserChallenge(String,Object)方法。同时，也能够集成现有的硬件令牌方案（比如 SafeWord 或 Radius）到 SRP 算法中。

第三步，创建 MBean，使得第二步完成的 SRPVerifierStore 接口实现能够借助于 JNDI 访问到，然后暴露需要的配置参数。本章内容除了给出默认实例 org.jboss.security.srp.SRPVerifierStoreService 外，后面还给出了基于 Java 属性文件的 SRPVerifierStore 实现。有了这两个实例，便能够实现安全性存储源的集成了。

8.5.2 深入 SRP 算法

SRP 算法的诱人之处在于能够使用简单的字符串密码实现客户和服务器的相互认证，而不需要安全通信渠道，对此开发者可能抱着怀疑的态度。JBossSX 实现的认证协议的序列图，如图 8-11 所示。

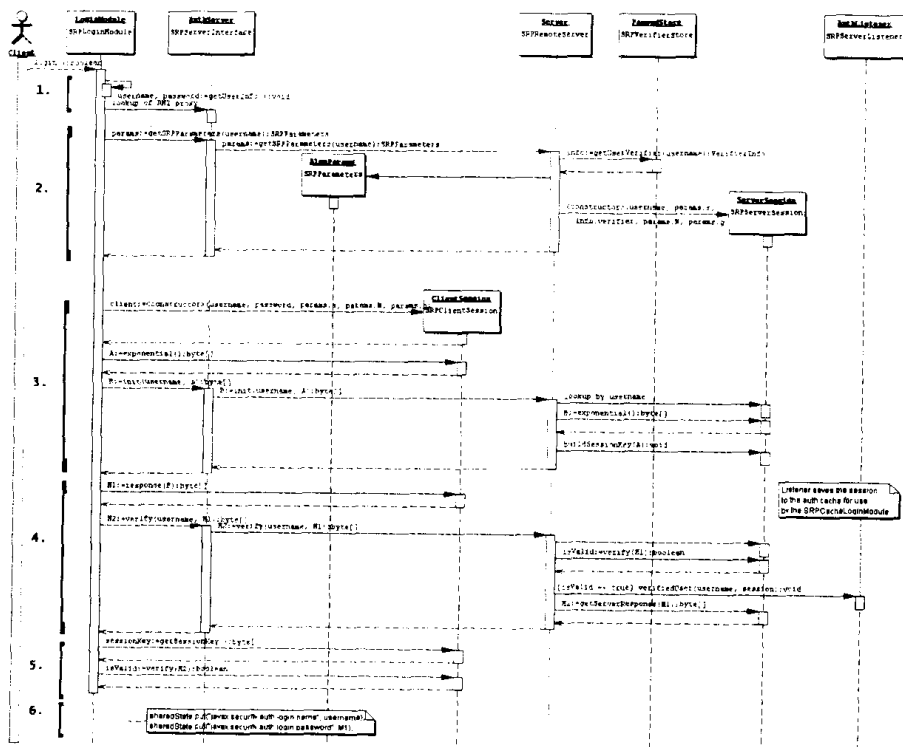


图 8-11 SRP 客户-服务器认证算法序列图

图 8-11 中给出的密钥消息交换的关键点将在下列内容中给出阐述。如果需要了解 SRP 算法的全部细节和理论，可以参考前面提到的 SRP 参考资料。为完成认证，需要经历 6

个步骤。

步骤

(1) 客户端 SRPLoginModule 从命名服务获得远程认证服务器的 SRPServerInterface 实例。

(2) 接下来, 客户端 SRPLoginModule 请求用户名关联的 SRP 参数, 以试图登录。当 SRP 算法初次将用户密码转换为验证器形式时, 需要从涉及 SRP 算法的大量参数中选择待使用的参数。JBossSX 实现没有硬编码参数 (最小的安全性风险), 而是将这部分参数信息作为交换协议的组成部分。使用 getSRPParameters(username) 能够获得特定用户名的 SRP 参数。

(3) 客户端 SRPLoginModule 启动 SRP 会话, 使用登录用户名、明文密码及步骤 2 获得的 SRP 参数创建 SRPClientSession 对象。然后, 客户创建随机数 A, 用于创建私有 SRP 会话密钥。紧接着, 客户调用 SRPServerInterface.init 方法初始化服务器端 SRP 会话, 并传入用户名和客户生成的随机数 A。服务器返回它创建的随机数 B, 从而完成公钥交换。

(4) 客户端 SRPLoginModule 获得上述消息交换生成的私有 SRP 会话密钥, 并将其保存下来作为登录 Subject 的私有凭证。通过调用 SRPClientSession.verify 方法能够验证步骤 (4) 的服务器响应 M2。如果成功, 则完成客户和服务器的相互认证。接下来, 客户端 SRPLoginModule 通过调用 SRPClientSession.response 方法, 并将服务器随机数 B 作为参数创建到服务器的请求 M1。借助于 SRPServerInterface.verify 方法能够将请求发送到服务器, 并将响应保存为 M2, 从而完成请求交换。此时, 服务器已经验证了该用户就是它所宣称的。

(5) 客户端 SRPLoginModule 将登录用户名和 M1 请求保存在 LoginModule 的共享状态 Map 中。标准 JBoss ClientLoginModule 将它们作为 Principal 名和凭证看待。M1 请求能够替换密码, 用于 J2EE 组件中方法调用的身份证明。其中, M1 请求是安全性极高的、关联 SRP 会话的散列。借助于第三方产品不能够截取用户的密码。

(6) 认证协议结束时, SRPServerSession 已经被放置在 SRPService 认证缓存中, 供 SRPCacheLoginModule 后续使用。

尽管 SRP 存在许多有趣的属性, 它仍然是 JBossSX 框架中有待改进的组件, 因为还存在一些缺陷, 如下所示:

- 由于 JBoss 将方法传输协议和完成认证操作的组件容器分离开, 所以未认证用户能够窃取到 SRP M1 请求, 并能够有效地使用该请求操作该用户名能够触发的任何任务。使用 SRP 会话密钥加密请求的自定义拦截器能够防止这种问题的出现。
- SRPService 维护了 SRP 会话缓存, 在经历了可配置时间后, 这些会话都将超时。一旦超时, 任何随后的 J2EE 组件访问都将失败, 因为当前不存在再次透明地获得 SRP 认证凭证的机制。因此, 或者将认证缓存超时设置为很大的值 (上限为 2 147 483 647 秒, 大概 68 年), 或者失败时在代码中处理再次认证。
- 默认时, 特定用户名只允许有一个 SRP 会话。由于获得的 SRP 会话创建了私有会话密钥, 即能够用于客户和服务器的加解密, 因此会话是有效的有状态对

象。从 JBoss 3.0.3 开始就新增了每用户多个 SRP 会话的能力，但是不允许使用某会话密钥加密数据，而用另一密钥解密数据。

为使用 J2EE 组件调用的端到端 SRP 认证，需要配置安全性域，其保护的组件需要使用 `org.jboss.security.srp.jaas.SRPCacheLoginModule`。其中，`SRPCacheLoginModule` 存在单个配置选项，即 `cacheJndiName`，其设置 SRP 认证 `CachePolicy` 实例的 JNDI 位置。它必须对应于 `SRPServicce MBean` 的 `AuthenticationCacheJndiName` 属性值。`SRPCacheLoginModule` 认证用户凭证，即从认证缓存中获得 `SRPServerSession` 对象的客户端请求，并将它与传入的请求进行比较。图 8-12 描述了 `SRPCacheLoginModule.login` 方法实现的操作。

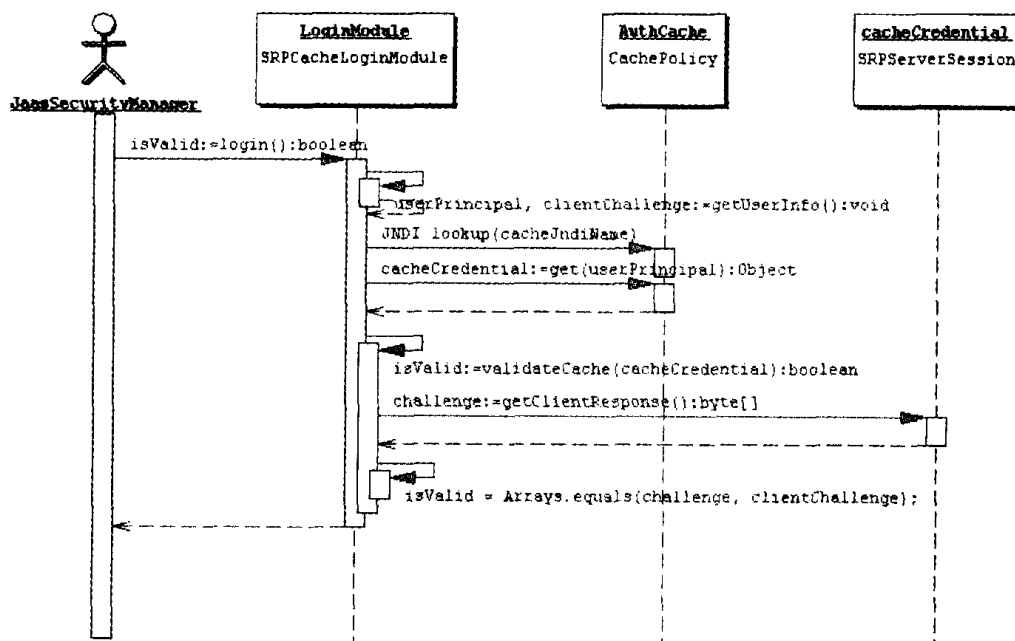


图 8-12 SRPCacheLoginModule 和 SRP 会话缓存交互的序列图

SRP 实例

前面已经给出了 SRP 的大量资料。至此，本文将给出实例，以证明实际场合中是如何使用 SRP 的。本实例借助于 SRP 完成用户的客户端认证，以及随后使用 SRP 会话请求作为用户凭证来访问受保护的简单 EJB。测试代码部署的 EJB jar 包含了 sar 存档，用于配置服务器端登录模块和 SRP 服务。同以前的实例一样，这里也使用 `SecurityConfig MBean` 动态安装服务器端登录模块配置。本实例还使用了 `SRPVerifierStore` 接口的自定义实现，即借助于 Java 属性文件，以完成内存内容的存取操作，而不是 `SRPVerifierStoreService` 使用的序列化对象存储文件。其中，自定义服务是 `org.jboss.chap8.ex3.service.PropertiesVerifierStore`。列表 8-15 给出了 jar 内容，其中含有 EJB 和 SRP 服务实例。

列表 8-15 chap8-ex3 jar 内容

```
[orb@toki examples]$ java -cp output/classes ListJar output/chap8/chap8-ex3.jar
output/chap8/chap8-ex3.jar
+- META-INF/MANIFEST.MF
+- META-INF/ejb-jar.xml
```

```

+- META-INF/jboss.xml
+- org/jboss/chap8/ex3/Echo.class
+- org/jboss/chap8/ex3/EchoBean.class
+- org/jboss/chap8/ex3/EchoHome.class
+- roles.properties+- users.properties
+- chap8-ex3.sar (archive)
| +- META-INF/MANIFEST.MF
| +- META-INF/jboss-service.xml
| +- META-INF/login-config.xml
| +- org/jboss/chap8/ex3/service/PropertiesVerifierStore$1.class
| +- org/jboss/chap8/ex3/service/PropertiesVerifierStore.class
| +- org/jboss/chap8/ex3/service/PropertiesVerifierStoreMBean.class
| +- org/jboss/chap8/service/SecurityConfig.class
| +- org/jboss/chap8/service/SecurityConfigMBean.class

```

本实例与 SRP 相关主要内容是 SRP MBean 服务配置和 SRP 登录模块配置。列表 8-16 给出了 chap8-ex3.sar 中的 jboss-service.xml 描述符，而列表 8-17 给出了客户端实例和服务端登录模块配置。

列表 8-16 用于 SRP 服务的 chap8-ex3.sar jboss-service.xml 描述符

```

<server>
  <!-- The custom JAAS login configuration that installs
  a Configuration capable of dynamically updating the
  config settings
  -->
  <mbean code="org.jboss.chap8.service.SecurityConfig"
  name="jboss.docs.chap8:service=LoginConfig-EX3">
    <attribute name="AuthConfig">META-INF/login-config.xml</attribute>
    <attribute
    name="SecurityConfigName">jboss.security:name=SecurityConfig</attribute>
  </mbean>

  <!-- The SRP service that provides the SRP RMI server and server side
  authentication cache -->
  <mbean code="org.jboss.security.srp.SRPService"
  name="jboss.docs.chap8:service=SRPService">
    <attribute name="VerifierSourceJndiName">srp-test/chap8-ex3</attribute>
    <attribute name="JndiName">srp-test/SRPServiceInterface</attribute>
    <attribute name="AuthenticationCacheJndiName">srp-test/
    AuthenticationCache</attribute>
    <attribute name="ServerPort">0</attribute>
    <depends>jboss.docs.chap8:service=PropertiesVerifierStore</depends>
  </mbean>

  <!-- The SRP store handler service that provides the user password verifier
  information -->

```

```
<mbean code="org.jboss.chap8.ex3.service.PropertiesVerifierStore"
name="jboss.docs.chap8.service=PropertiesVerifierStore">
  <attribute name="JndiName">srp-test/chap8-ex3</attribute>
</mbean>
</server>
```

列表 8-17 chap8-ex3 客户端和服务端 SRP 登录模块配置

```
// Client side standard JAAS configuration fragment
srp {
org.jboss.security.srp.jaas.SRPLoginModule required
srpServerJndiName="srp-test/SRPServiceInterface"
;
org.jboss.security.ClientLoginModule required
password-stacking="useFirstPass"
;
};

// Server side XMLLoginConfig configuration fragment
<application-policy name = "chap8-ex3">
  <authentication>
    <login-module code = "org.jboss.security.srp.jaas.SRPCacheLoginModule"
      flag = "required">
      <module-option name = "cacheJndiName">srp-test/AuthenticationCache
    </module-option>
    </login-module>
    <login-module code =
      "org.jboss.security.auth.spi.UsersRolesLoginModule" flag = "required">
      <module-option name = "password-stacking">useFirstPass
    </module-option>
    </login-module>
  </authentication>
</application-policy>
```

其中，实例使用了 ServiceConfig、PropertiesVerifierStore 及 SRPService MBean 服务。请注意，PropertiesVerifierStore 的 JndiName 属性等于 SRPService 的 VerifierSourceJndiName 属性，SRPService 依赖于 PropertiesVerifierStore。由于 SRPService 需要使用 SRPVerifierStore 接口实现访问用户密码验证信息，因此上述需求是必须满足的。

列表 8-17 给出的客户端登录模块配置使用了 SRPLoginModule，其提供的 srpServerJndiName 选项值对应于 JBoss 服务器组件 SRPService 的 JndiName 属性值（“srp-test/SRPServiceInterface”）。另外，开发者还需要配置 ClientLoginModule，并为它提供 password-stacking="userFirstPass" 值，从而能够将 SRPLoginModule 生成的用户认证凭证委派给 EJB 调用层。

有关服务器端登录模块配置有两点值得开发者注意。其一，配置选项 cacheJndiName=srp-test/AuthenticationCache，以告知 SRPCacheLoginModule 登录模块 CachePolicy 的位置。

其中,该 CachePolicy 含有 SRPService 认证用户的 SRPServiceSession。该位置对应于 SRPService AuthenticationCacheIndiName 属性值。其二,该配置包含了 UsersRolesLoginModule 登陆模块,并带有配置选项 password-stacking="useFirstPass"。这是 SRPCacheLoginModule 使用的第二个登录配置,因为 SRP 只是认证技术。UsersRolesLoginModule 接受的凭证由 SRPCacheLoginModule 验证,从而能够设置 Principal 的角色以决定 Principal 的许可。UsersRolesLoginModule 使用基于属性文件的授权辅助 SRP 认证。EJB jar 包括的 roles.properties 含有用户角色。

开发者可以从光盘目录运行如下命令行,以执行 ex3。

```
[starksm@banshee examples]$ ant -Dchap=chap8 -Dex=3 run-example
Buildfile: build.xml
...
run-example3:
[copy] Copying 1 file to /tmp/jboss-3.2.3/server/default/deploy
[echo] Waiting for 5 seconds for deploy...
[java] Logging in using the 'srp' configuration
[java] Created Echo
[java] Echo.echo()#1 = This is call 1
[java] Echo.echo()#2 = This is call 2
```

目录 examples/logs 下有 ex3-trace.log 文件。这是 SRP 算法的客户端执行具体细节,展示了公钥、请求、会话密钥以及验证的具体步骤。



相对于其他简单实例而言,这里的客户运行时间较长,原因在于构建客户的公钥。其中,涉及到创建安全性极高的随机数,并且这个过程初次出现时需要花费较多时间。如果在同一 VM 中登出,然后再登入,则该过程会很快。另外请注意,“Echo.echo()#2”因出现了认证异常,而告失败。客户在完成初次调用后,等待了 15 秒,以证明 SRPService 缓存过期行为。因此,SRPService 缓存策略超时只设置了 10 秒。正如前面提到的一样,需要将缓存超时设为很大的数值,或者在出现失败时程序处理再次认证。

8.6 使用 Java 2 安全性管理器运行 JBoss

默认情况下,JBoss 服务器并没有使用 Java 2 安全性管理器。如果需要使用 Java 2 许可约束代码特权,则需要配置 JBoss 服务器在安全性管理器下运行,即通过配置 JBoss 服务器发布版 bin 目录 run.bat 或 run.sh 脚本中的 JVM 选项。必须提供的两个 JVM 选项如下:

- **java.security.manager:** 如果没有指定任何默认安全性管理器,则可以使用该选项,这也是推荐的安全性管理器。同时,也可以为 java.security.manager 选项指定值,即自定义安全性管理器实现。该值必须是 java.lang.SecurityManager 子类的全限定类名,从而能够告知策略文件应该扩充 JVM 安装配置的默认安全性策略。
- **java.security.policy:** 用于指定策略文件,辅助 JVM 提供默认安全性策略信息。开发者可以使用如下两种形式给出该选项: java.security.policy=policyFileURL 或

java.security.policy==policyFileURL。第一种形式指定，策略文件应该扩充 JVM 安装配置的默认安全性策略。第二种形式指定，仅使用给出的策略文件。其中，policyFileURL 值可能是协议处理器中允许的任何 URL，或者文件路径。

列表 8-18 给出了用于 Win32 的 run.bat 启动脚本的片段。其中，添加了这两个选项以启动 JBoss。

列表 8-18 用于 Win32 的 run.bat 启动脚本片段（使用 Java 2 安全性管理器运行 JBoss）

```
...

set CONFIG=%1
@if "%CONFIG%" == "" set CONFIG=default
set PF=../conf/%CONFIG%/server.policy
set OPTS=-Djava.security.manager
set OPTS=%OPTS% -Djava.security.policy=%PF%
echo JBOSS_CLASSPATH=%JBOSS_CLASSPATH%
java %JAXP% %OPTS% -classpath "%JBOSS_CLASSPATH%" org.jboss.Main %*
```

列表 8-19 给出了用于 UNIX/Linux 的 run.sh 启动脚本的片段。其中，添加了这两个选项以启动 JBoss。

列表 8-19 用于 UNIX/Linux 的 run.sh 启动脚本片段（使用 Java 2 安全性管理器运行 JBoss）

```
...

CONFIG=$1
if [ "$CONFIG" == "" ]; then CONFIG=default; fi
PF=../conf/$CONFIG/server.policy
OPTS=-Djava.security.manager
OPTS="$OPTS -Djava.security.policy=$PF"
echo JBOSS_CLASSPATH=$JBOSS_CLASSPATH
java $HOTSPOT $JAXP $OPTS -classpath $JBOSS_CLASSPATH org.jboss.Main $@
```

上述两个脚本都将安全性策略文件指定为 JBoss 配置文件集合目录中的 server.policy 文件。其中，JBoss 配置文件集合目录对应于传入脚本的第一个参数，即配置名。从而，只需要为每个配置文件集合维护单个安全性策略文件，而不需要修改启动脚本。

生效 Java 2 安全性是很简单的事情，但 Java 2 安全性的难点在于建立允许的许可。如果打开默认配置文件集合中的 server.policy 文件，将看到的许可语句如下：

```
grant {
    // Allow everything for now
    permission java.security.AllPermission;
};
```

它有效地将安全性许可检查所有代码失效，即宣告任何代码能够做任何事情。当然，这不是很理智的事情。开发者需要创建适合自身的许可。

当前 JBoss 提供的具体 java.lang.RuntimePermission 集合如下，如表 8-1 所示。

表 8-1 具体 java.lang.RuntimePermission 集合

目 标 名	许 可	风 险
org.jboss.security.SecurityAssociation.getPrincipalInfo	访问 org.jboss.security.SecurityAssociation getPrincipal() 和 getCredentials 方法	能够查看当前线程调用者和凭证
org.jboss.security.SecurityAssociation.setPrincipalInfo	访问 org.jboss.security.SecurityAssociation setPrincipal() 和 setCredentials 方法	能够设置当前线程调用者和凭证
org.jboss.security.SecurityAssociation.setServer	访问 org.jboss.security.SecurityAssociation setServer 方法	能够生效或失效调用者 Principal 和凭证的多线程存储
org.jboss.security.SecurityAssociation.setRunAsRole	访问 org.jboss.security.SecurityAssociation pushRunAsRole 和 popRunAsRole 方法	能够改变当前调用者 run-as 角色 Principal

为总结本次讨论，下面介绍一些调试安全性策略设置时鲜为人知的小技巧。使用 JVM 提供的各种调试标志，开发者能够掌握安全管理器是如何使用安全性策略文件和许可使用了哪些策略文件。以如下方式运行 JVM 能够查看到所有调试标志：

```
[nr@toki bin]$ java -Djava.security.debug=help
```

```
all          turn on all debugging
access       print all checkPermission results
combiner     SubjectDomainCombiner debugging
jar          jar verification
logincontext login context results
policy       loading and granting
provider     security provider debugging
scl          permissions SecureClassLoader assigns
```

The following can be used with access:

```
stack       include stack trace
domain      dumps all domains in context
failure     before throwing exception, dump stack and domain that didn't have permission
```

Note: Separate multiple options with a comma

开发者使用-Djava.security.debug=all，能够提供最详细的输出信息，但是输出内容太丰富了。因此，如果开发者还没有理解某个具体安全性失败，则这里提供了一种很好的办法。使用-Djava.security.debug=access,failure 能够提供比 all 模式更少的信息，以助于调试许可失败。虽然上述方法还是输出了很多信息，但同 all 模式相比还是不错的，因为只有出现访问失败时才会输出安全域信息。

8.7 使用 JSSE 为 JBoss 提供 SSL

JBoss 使用 Java 安全套接字扩展（Java Secure Socket Extension, JSSE）。JBoss 绑定了 JSSE, JSSE 来自 JDK 1.4。更多 JSSE 信息，请参考 JSSE 主页 <http://java.sun.com/products/jsse/>

index.html。接下来做一个简单测试，看看 JBoss 绑定的 JSSE 是否能够使用。

```
import java.net.*;
import javax.net.ServerSocketFactory;
import javax.net.ssl.*;

public class JSSE_install_check
{
    public static void main(String[] args) throws Exception
    {
        Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
        ServerSocketFactory factory =
            SSLServerSocketFactory.getDefault();
        SSLServerSocket sslSocket = (SSLServerSocket)
            factory.createServerSocket(12345);

        String [] cipherSuites = sslSocket.getEnabledCipherSuites();
        for(int i = 0; i < cipherSuites.length; i++)
        {
            System.out.println("Cipher Suite " + i +
                " = " + cipherSuites[i]);
        }
    }
}
```

本书实例中附带了一个测试用例，使用如下命令行能够运行它。如果使用 -Djava.security.debug=all 选项，将输出大量信息。

```
[nr@toki examples]$ ant -Dchap=chap8 -Dex=4a run-example
Buildfile: build.xml
...
run-example4a:
[echo] Testing JSSE availability
[java] keyStore is :
[java] keyStore type is : jks
[java] init keystore
[java] init keymanager of type SunX509
[java] trustStore is: /System/Library/Frameworks/JavaVM.framework/Versions/1.4.2/Home/lib/
security/cacerts
[java] trustStore type is : jks
[java] init truststore
...
[java] init context
[java] trigger seeding of SecureRandom
[java] done seeding SecureRandom
[java] Cipher Suite 0 = SSL_RSA_WITH_RC4_128_MD5
[java] Cipher Suite 1 = SSL_RSA_WITH_RC4_128_SHA
```

```
[java] Cipher Suite 2 = TLS_RSA_WITH_AES_128_CBC_SHA
[java] Cipher Suite 3 = TLS_DHE_RSA_WITH_AES_128_CBC_SHA
[java] Cipher Suite 4 = TLS_DHE_DSS_WITH_AES_128_CBC_SHA
[java] Cipher Suite 5 = SSL_RSA_WITH_3DES_EDE_CBC_SHA
[java] Cipher Suite 6 = SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
[java] Cipher Suite 7 = SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
[java] Cipher Suite 8 = SSL_RSA_WITH_DES_CBC_SHA
[java] Cipher Suite 9 = SSL_DHE_RSA_WITH_DES_CBC_SHA
[java] Cipher Suite 10 = SSL_DHE_DSS_WITH_DES_CBC_SHA
[java] Cipher Suite 11 = SSL_RSA_EXPORT_WITH_RC4_40_MD5
[java] Cipher Suite 12 = SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
[java] Cipher Suite 13 = SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
[java] Cipher Suite 14 = SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
```

JSSE jar 包括 JBOSS_DIST/client 目录中的 jcert.jar、jnet.jar 及 jsse.jar。

一旦测试了上述 JSSE 应用，则需要提供 X509 证书形式的公钥/私钥对供 SSL 服务器 Socket 使用。本文使用 JDK 1.3 keytool 为该实例创建了自签名证书，并将获得的 keystore 文件（chap8.keystore）包括在 chap8 源代码目录中。其创建过程如下：

```
[nr@toki examples]$ keytool -genkey -alias rmi+ssl -keyalg RSA -keystore chap8.keystore -validity
3650
[orb@toki examples]$ keytool -genkey -alias rmi+ssl -keyalg RSA -keystore chap8.keystore -validity
3650
Enter keystore password: rmi+ssl
What is your first and last name?
[Unknown]: Chapter 8 SSL Example
What is the name of your organizational unit?
[Unknown]: JBoss Book
What is the name of your organization?
[Unknown]: JBoss, Inc.
What is the name of your City or Locality?
[Unknown]: Issaquah
What is the name of your State or Province?
[Unknown]: WA
What is the two-letter country code for this unit?
[Unknown]: US
Is CN=Chapter 8 SSL Example, OU=JBoss Book, O="JBoss, Inc.", L=Issaquah, ST=WA, C=US correct?
[no]: yes

Enter key password for <rmi+ssl>
(RETURN if same as keystore password):
```

这将生成 keystore 文件（chap8.keystore）。keystore 存储了安全性密钥。在 keystore 中存在两种不同类型的入口：

- 密钥入口：每个入口持有非常敏感的加密密钥信息，使用了受保护格式存储防止非法访问。通常，存储在这种入口中的密钥是加密密钥，或者私钥。其中，证书

“链”在提供私钥的同时都会存在相应的公钥。keytool 和 jarsigner 工具仅仅处理私钥及其关联的证书链。

- 可信任证书入口：每个入口含有单个属于第三方的公钥证书。称之为“可信任证书”的理由在于，keystore 持有人确信证书中的公钥确实属于证书持有人（“属主”）标识的身份。证书发行者通过签署证书能够保证它是可信任的。

使用 keytool 工具列举 src/main/org/jboss/chap8/chap8.keystore 文件内容，即显示自签名证书过程如下：

```
[nr@toki examples]$ keytool -list -v -keystore src/main/org/jboss/chap8/chap8.keystore
Enter keystore password: rmi+ssl

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: rmi+ssl
Creation date: Nov 8, 2001
Entry type: keyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Chapter8 SSL Example, OU=JBoss Book, O="JBoss Group, LLC", L=Issaquah, ST=WA, C=US
Issuer: CN=Chapter8 SSL Example, OU=JBoss Book, O="JBoss Group, LLC", L=Issaquah, ST=WA, C=US
Serial number: 3beb5271
Valid from: Thu Nov 08 21:50:09 CST 2001 until: Sun Nov 06 21:50:09 CST 2011
Certificate fingerprints:
MD5: F6:1B:2B:E9:A5:23:E7:22:B2:18:6F:3F:9F:E7:38:AE
SHA1: F2:20:50:36:97:86:52:89:71:48:A2:C3:06:C8:F9:2D:F7:79:00:36

*****
*****
```

在确认 JBoss 服务器能够使用 JSSE 及存在带有证书的 keystore 后，本文可以开始为 JBoss EJB 访问配置 SSL，即通过配置 EJB Invoker 的 RMI 套接工厂。JBossSX 框架提供了如下接口实现，即 java.rmi.server.RMIServerSocketFactory 和 java.rmi.server.RMIClientSocket Factory，从而能够在 SSL 加密套接字上使用 RMI。上述接口的实现类分别是 org.jboss.security.ssl.RMISSLServerSocketFactory 和 org.jboss.security.ssl.RMISSLClientSocket Factory。通过两个步骤能够完成在 SSL 基础上使用 RMI 访问 EJB。

第一步，将 keystore 作为 SSL 服务器认证的数据源，即通过 org.jboss.security.plugins.JaasSecurityDomain MBean 配置完成。列表 8-20 给出了 chap8/ex4 目录下包括 JaasSecurity Domain 定义的 jboss-service.xml 描述符。

列表 8-20 用于 RMI/SSL 的 JaasSecurityDomain 配置实例

```
<!-- The SSL domain setup -->
```

```

<mbean code="org.jboss.security.plugins.JaasSecurityDomain"
  name="jboss.security:service=JaasSecurityDomain,domain=RMI+SSL">
  <constructor>
    <arg type="java.lang.String" value="RMI+SSL"/>
  </constructor>
  <attribute name="KeyStoreURL">chap8.keystore</attribute>
  <attribute name="KeyStorePass">rmi+ssl</attribute>
</mbean>

```

JaasSecurityDomain 是标准 JaasSecurityManager 的子类，并添加了 keystore、JSSE KeyManagerFactory 及 TrustManagerFactory 类的使用。它扩展了基本的安全性管理器，以支持 SSL 和其他要求安全性密钥的加密操作。本配置简单地使用指定密码，从实例 4 MBeansar 中装载 chap8.keystore。

第二步，使用支持 SSL 的 JBossSX RMI 套接字工厂定义 EJB Invoker 配置。其中，需要为 JRMPInvoker 定义自定义配置和设置使用了该 Invoker 的 EJB。在第 5 章“JBoss 之 EJB——EJB 容器配置和架构”中给出了自定义配置的详细介绍。列表 8-21 给出了在 SSL 基础上借助 RMI 访问无状态会话 Bean 的配置实例。列表上半部分是 jboss-service.xml 描述符，定义自定义 JRMPInvoker；下半部分给出了使用了 SSL Invoker 的实例 4 “EchoBean4”配置。下面将在无状态会话 Bean 实例中使用该配置。

列表 8-21 jboss-service.xml 和 jboss.xml 配置（使用 SSL 的实例 4 无状态会话 Bean）

```

// The jboss-service.xml SSL JRMPInvoker MBean Configuration
<mbean code="org.jboss.invocation.jrmp.server.JRMPInvoker"
  name="jboss:service=invoker,type=jrmp,socketType=SSL">
  <attribute name="RMIObjectPort">4445</attribute>
  <attribute name="RMIClientSocketFactory">
    org.jboss.security.ssl.RMISSLClientSocketFactory
  </attribute>
  <attribute name="RMIServerSocketFactory">
    org.jboss.security.ssl.RMISSLServerSocketFactory
  </attribute>
  <attribute name="SecurityDomain">java:/jaas/RMI+SSL</attribute>
  <depends>jboss.security:service=JaasSecurityDomain,domain=RMI+SSL
  </depends>
</mbean>

// The jboss.xml session bean configuration to use the SSL invoker
<?xml version="1.0"?>
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>EchoBean4</ejb-name>
      <configuration-name>Standard Stateless SessionBean</configuration-name>
      <home-invoker>jboss:service=invoker,type=jrmp,socketType=SSL
      </home-invoker>
    </session>
  </enterprise-beans>
</jboss>

```

```
<bean-invoker>jboss:service=invoker,type=jmp,socketType=SSL
</bean-invoker>
</session>
</enterprise-beans>
</jboss>
```

相应的源代码通过 `src/main/org/jboss/chap8/ex4` 目录能够找得到。另外，本文还提供了一个简单无状态会话 Bean 实例，它含有一个返回输入参数的 `echo` 方法。其实，很难判断 SSL 是否起作用了，除非调用失败。因此，下面提供了两种方式运行实例 4 客户端，以证明 EJB 确实是使用了 SSL。如下所示，首先使用默认配置文件集合启动了 JBoss 服务器，然后运行实例 4b。

```
[nr@toki examples]$ ant -Dchap=chap8 -Dex=4b run-example
Buildfile: build.xml
...
run-example4b:
[copy] Copying 1 file to /tmp/jboss-3.2.3/server/default/deploy
[echo] Waiting for 15 seconds for deploy...
[java] Exception in thread "main" java.rmi.ConnectIOException: error during JRMP connection
establishment; nested exception is:
[java] javax.net.ssl.SSLHandshakeException: sun.security.validator.ValidatorException: No trusted
certificate found
[java] at sun.rmi.transport.tcp.TCPChannel.createConnection(TCPChannel.java:274)
[java] at sun.rmi.transport.tcp.TCPChannel.newConnection(TCPChannel.java:171)
[java] at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:101)
[java] at org.jboss.invocation.jmp.server.JRMPInvoker_Stub.invoke(Unknown Source)
[java] at org.jboss.invocation.jmp.interfaces.JRMPInvokerProxy.invoke(JRMPInvokerProxy.
java:135)
[java] at org.jboss.invocation.InvokerInterceptor.invoke(InvokerInterceptor.java:96)
[java] at org.jboss.proxy.TransactionInterceptor.invoke(TransactionInterceptor.java:46)
[java] at org.jboss.proxy.SecurityInterceptor.invoke(SecurityInterceptor.java:45)
[java] at org.jboss.proxy.ejb.HomeInterceptor.invoke(HomeInterceptor.java:173)
[java] at org.jboss.proxy.ClientContainer.invoke(ClientContainer.java:85)
[java] at $Proxy0.create(Unknown Source)
[java] at org.jboss.chap8.ex4.ExClient.main(ExClient.java:31)
[java] Caused by: javax.net.ssl.SSLHandshakeException: sun.security.validator.ValidatorException:
No trusted certificate found
[java] at com.sun.net.ssl.internal.ssl.BaseSSLSocketImpl.a(DashoA6275)
...
[java] Caused by: sun.security.validator.ValidatorException: No trusted certificate found
[java] at sun.security.validator.SimpleValidator.buildTrustedChain(SimpleValidator.java:304)
[java] at sun.security.validator.SimpleValidator.engineValidate(SimpleValidator.java:107)
[java] at sun.security.validator.Validator.validate(Validator.java:202)
[java] at com.sun.net.ssl.internal.ssl.X509TrustManagerImpl.checkServerTrusted(DashoA6275)
[java] at com.sun.net.ssl.internal.ssl.JsseX509TrustManager.checkServerTrusted(DashoA6275)
[java] ... 22 more
```

```
[java] Java Result: 1
```

```
BUILD SUCCESSFUL
```

```
Total time: 19 seconds
```

运行结果和预期相同，这也是实例 4b 的目的。请注意，列表给出的异常输出信息是经过裁减的，因此和实际的输出有些出入。列表给出的异常信息很清楚地给出了该实例在使用 Sun JSSE 类同 JBoss EJB 容器通信。另外，从异常信息还可以看出，JBoss 服务器证书（即自签名证书）的有效性不能够由默认证书授权中心的任何证书签署。这和预期的一样，因为同 JSSE 包一起发布的默认证书授权中心只包括了知名度非常高的授权中心，比如 VeriSign、Thawte 及 RSA Data Security。为了使 EJB 客户接受所提供的自签名证书是有效的，开发者需要告知 JSSE 类使用 chap8.keystore 作为其 truststore。truststore 也是 keystore，只不过它含有用于签署其他证书的公钥证书。因此使用 -Dex=4，而不是 -Dex=4b 运行实例 4，并使用 javax.net.ssl.trustStore 系统属性传入正确的 truststore。

```
[nr@toki examples]$ ant -Dchap=chap8 -Dex=4 run-example
Buildfile: build.xml
...
run-example4:
[copy] Copying 1 file to /tmp/jboss-3.2.3/server/default/deploy
[echo] Waiting for 5 seconds for deploy...
[java] 1 [HandshakeCompletedNotify-Thread] DEBUG org.jboss.security.ssl.RMISSSLClientSocket Factory
- SSL handshakeCompleted, cipher=SSL_RSA_WITH_RC4_128_MD5, peerHost=127.0.0.1
[java] Created Echo
[java] Echo.echo()#1 = This is call 1

BUILD SUCCESSFUL
Total time: 15 seconds
```

通过“SSL handshakeCompleted”能看出，这次运行只使用到 SSL Socket。这也是 RMISSSLClientSocketFactory 类的调试输出信息。如果没有将客户端配置成能够打印 Log4j debug 级信息的，则很难看出使用了 SSL。如果注意到运行时间和系统 CPU 的负载情况，则一定存在细微的区别。SSL 同 SRP 一样，涉及到使用了安全性极高的随机数，因此初次使用时需要花费较长时间。从而，能够看到高 CPU 使用和启动时间。

另外，如下的异常也是由于上述使用 SSL 引起的。如果实例运行的目标机器很慢，则运行实例 4b 时，会出现如下的类似异常：

```
javax.naming.NameNotFoundException: EchoBean4 not bound
    at sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer
    at sun.rmi.transport.StreamRemoteCall.executeCall
    at sun.rmi.server.UnicastRef.invoke
    at org.jnp.server.NamingServer_Stub.lookup
    at org.jnp.interfaces.NamingContext.lookup
    at org.jnp.interfaces.NamingContext.lookup
    at javax.naming.InitialContext.lookup
    at org.jboss.chap8.ex4.ExClient.main(ExClient.java:29)
```



```
Exception in thread "main"
```

```
Java Result: 1
```

问题在于客户发出访问请求时，JBoss 服务器还未部署完成 EJB 实例，原因在于 SSL 服务器 Socket 使用的安全随机数生成器还未完成其初次设置。如果遇到这种问题，请重新运行实例，或者在 chap8 build.xml Ant 脚本中增加部署等待时间。

8.8 配置用于防火墙后的 JBoss

JBoss 运行时，启动了很多基于 Socket 的服务，并打开了监听端口。本节内容列举出这些打开端口的服务，从而为配置访问防火墙后的 JBoss 作准备。表 8-2 给出了 default 配置文件集合中给出了端口、Socket 类型、相关服务及配置这些服务的连接。表 8-3 给出了 all 配置文件集合中，使用了其他端口的类似信息。

表 8-2 default 配置中的端口

端 口	类 型	服 务	参 考
1099	TCP	org.jboss.naming.NamingService	3.2 “JBossNS 架构”
1098	TCP	org.jboss.naming.NamingService	3.2 “JBossNS 架构”
1162	UDP	org.jboss.jmx.adaptor.snmp.trapd.TrapdService	2.6.2 中的“Trap 服务事件”
4444	TCP	org.jboss.invocation.jrmp.server.JRMPInvoker	2.7.2 中的“JRMPInvoker-RMI/JRMP 传输”
4445	TCP	org.jboss.invocation.pooled.server.PooledInvoker	2.7.2 中的“PooledInvoker-RMI/套接字传输”
8009	TCP	org.jboss.web.tomcat.tc4.EmbeddedTomcatSevice	9.2.1 中的“Connector”
8080	TCP	org.jboss.web.tomcat.tc4.EmbeddedTomcatSevice	9.2.1 中的“Connector”
8083	TCP	org.jboss.web.WebService	10.6 “RMI 动态类装载”
8090	TCP	org.jboss.mq.il.oil.OILServerILService	6.3.3 “org.jboss.mq.il.oil.OILServerILService（已丢弃）”
8092	TCP	org.jboss.mq.il.oil2.OIL2ServerILService	无
8093	TCP	org.jboss.mq.il.uil2.UILServerILService	6.3.5 “org.jboss.mq.il.uil2.UILServerILService”
0 ^a	TCP	org.jboss.mq.il.rmi.RMIServerILService	6.3.2 “org.jboss.mq.il.rmi.RMIServerILService（已丢弃）”
0 ^b	UDP	org.jboss.jmx.adaptor.snmp.agent.SnmpAgentService	2.6.1 “SNMP 适配器服务”

注：a. 该服务绑定到匿名 TCP 端口，并且不支持端口配置或绑定接口。

b. 该服务绑定到匿名 UDP 端口，并且不支持端口配置或绑定接口。

表 8-3 all 配置中的其他端口

端 口	类 型	服 务	参 考
1100	TCP	org.jboss.ha.jndi.HANamingService	3.2 中的“群集环境中的命名查找”
0 ^a	TCP	org.jboss.ha.jndi.HANamingService	3.2 中的“群集环境中的命名查找”
1102	UDP	org.jboss.ha.jndi.HANamingService	3.2 中的“群集环境中的命名查找”
3528	TCP	org.jboss.invocation.iiop.IIOPInvoker	2.7.2 中的“IIOPInvoker-RMI/IIOP 传输”

(续表)

端 口	类 型	服 务	参 考
45566 ^b	UDP	org.jboss.ha.framework.server.ClusterPartition	参考《Clustering Guide》一书

注: a. 当前是匿名端口, 但可以通过 RmiPort 属性设置。

b. 两个附加 UDP 端口。一个使用 rcv_port 设置, 另一个不能。

8.9 如何保护 JBoss 服务器

用户需要保护或删除随 JBoss 发布的若干个管理访问应用, 从而能够防止部署环境中管理功能的非法访问。本节描述不同的管理服务, 并阐述如何保护它们。

8.9.1 jmx-console.war

deploy 目录中的 jmx-console.war 提供了 JMX 微内核的 HTML 视图。因此, 它提供了访问管理功能, 比如关闭服务器、停止服务及部署新服务等。同其他 Web 应用一样, 需要保护或者删除它。

8.9.2 web-console.war

deploy/management 目录中的 web-console.war 是另一个 Web 应用, 它提供 JMX 微内核的 HTML 视图。它使用 Applet 和 HTML 视图, 提供了同 jmx-console.war 相同的管理功能。因此, 应该保护或删除它。web-console.war 的 WEB-INF/web.xml 和 WEB-INF/jboss-web.xml 中含有安全性配置模板, 不过默认时并没有启用。

8.9.3 http-invoker.sar

deploy 目录中的 http-invoker.sar 服务为 EJB 和 JNDI 命名服务提供了 RMI/HTTP 访问方式。它包括的 Servlet 将调用请求分发到 MBeanServer 中。其中, 这里的调用请求是已压包的 org.jboss.invocation.Invocation 对象类型。由于用户能够格式化合适的 HTTP POST 请求, 因此借助于 HTTP 能够有效地访问支持分离式 Invoker 操作的 MBean。为了保护该服务, 用户需要保护 JMXInvokerServlet Servlet, 通过 http-invoker.sar/invoker.war/WEB-INF/web.xml 描述符能够找到。在默认情况下, 该 sar 提供了用于/restricted/JMXInvokerServlet 路径的保护性映射。具体做法是首先删除其他的路径, 然后配置 http-invoker 安全性域, 即修改 http-invoker.sar/invoker.war/WEB-INF/jboss-web.xml 描述符。

8.9.4 jmx-invoker-adaptor-server.sar

jmx-invoker-adaptor-server.sar 服务借助于 RMI 兼容接口暴露了 JMX MBeanServer 接口。其中, 使用 RMI/JRMP 分离式 Invoker 服务能够获得这些 RMI 兼容接口。目前, 保护该服务的惟一途径是将协议换成 RMI/HTTP, 然后保护上面提及的 http-invoker.sar。JBoss 的后续版本将把它部署成带有安全性拦截器的 XMBBean, 从而能够支持基于角色的访问检查。如果用户倾向于该服务, 在 2.4.3 中的“版本 3, 为 JNDIMap XMBBean 添加安全性和远程访问”小节内 XMBBean 实例中给出了具体设置步骤。

第9章 集成 Servlet 容器

本章讨论集成第三方 Web 容器到 JBoss 应用服务器框架的具体步骤。Web 容器是 J2EE 服务器组件，使用它能够实现对 Servlet 和 JSP 的访问。Tomcat 是使用最为广泛的 Servlet 容器，它也是 JBoss 使用的默认 Web 容器。

集成 Servlet 容器到 JBoss 包含两个步骤。第一，使用可选 jboss-web.xml 描述符将 web.xml JNDI 信息映射到 JBoss JNDI 命名空间中。第二，将认证和授权操作委派给 JBoss 安全性层。现存的 `org.jboss.web.AbstractWebContainer` 能够简化上述任务。本章前半部分重点阐述如何使用 `AbstractWebContainer` 类集成 Web 容器。后半部分，本章介绍若干配置主题，比如如何为 JBoss/Tomcat 绑定配置使用安全套接字层（Secure Socket Layer, SSL）和 Apache。

9.1 AbstractWebContainer 类

为将 Web 容器集成到 JBoss 中，`org.jboss.web.AbstractWebContainer` 类提供了模板模式实现。如果 Web 容器供应商希望集成各自的 Web 容器到 JBoss 服务器中，则他们必须创建 `AbstractWebContainer` 的子类，并提供具体 Web 容器的配置和 WAR 部署步骤。`AbstractWebContainer` 能够分析标准 J2EE 的 `web.xml`，即 Web 应用部署描述符中的 JNDI 和安全性元素，以及分析 JBoss 的 `jboss-web.xml` 描述符。为生成集成的 JNDI 环境和安全性上下文，需要分析这些部署描述符。本书在其他章节中已给出了 `jboss-web.xml` 描述符的大部分元素。图 9-1 给出了 `jboss-web.xml` 描述符的 DTD 概述。通过 `JBoss_DIST/docs/dtd` 能够找到完整的带注释 DTD。

本文还有两个元素还没讨论，即 `context-root` 和 `virtual-host`。`context-root` 元素允许指定定位 Web 应用的前缀。如果 WAR 部署成单独 Web 应用，则可以使用 `jboss-web.xml` 中的 `context-root` 设置 WAR 根路径；如果 WAR 部署成 EAR 中的 Web 应用，则需要使用 EAR `application.xml` 描述符的 `context-root` 元素设置根路径，而不能使用上述 `context-root` 元素。列表 9-1 给出的 `jboss-web.xml` 描述符实例，它演示了如何将 WAR 映射到根上下文。

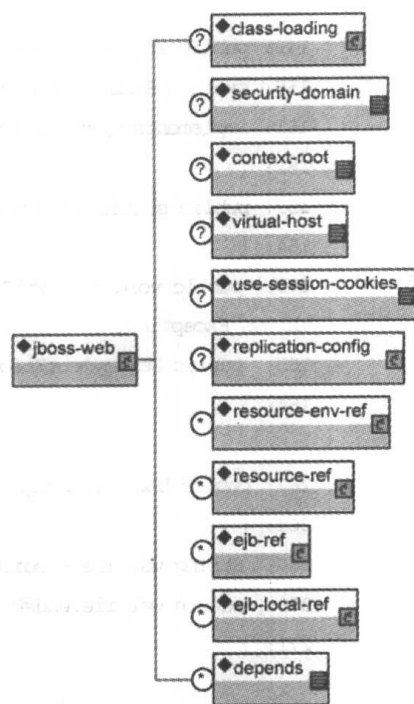


图 9-1 jboss-web.xml 描述符的完整 DTD

列表 9-1 jboss-web.xml 描述符实例（映射 WAR 到根上下文）

```
<jboss-web>
  <!-- An empty context root map the war to the root context,
       e.g., http://localhost:8080/ -->
  <context-root />
</jboss-web>
```

virtual-host 元素指定部署 Web 应用的虚拟主机的 DNS 名。设置 Servlet 上下文的虚拟主机的具体细节需要取决于各个特定 Servlet 容器。本章后续讨论 Tomcat Servlet 容器时，开发者将会看到 virtual-host 元素的使用实例。

9.1.1 AbstractWebContainer 契约

抽象 AbstractWebContainer 实现了 org.jboss.web.AbstractWebContainerMBean 接口。JBoss J2EE 部署器需要使用它，以委派 WAR 文件的安装任务。列表 9-2 给出了 AbstractWebContainer 的主要方法。

列表 9-2 AbstractWebContainer 的主要方法

```
149: public abstract class AbstractWebContainer
150: extends SubDeployerSupport
151: implements AbstractWebContainerMBean
152: {
153: public static interface WebDescriptorParser
154: {
155:
156:
157:
158:
159:
160:
161:
162:
163:
164:
165:
166:
167:
168:
169:
170:
171:
172:
173:
174:
175: public void parseWebAppDescriptors(ClassLoader loader, WebMetaData metaData)
176: throws Exception;
177:
178:
179:
180:
181:
182:
183: public DeploymentInfo getDeploymentInfo();
184: }

267: public boolean accepts(DeploymentInfo sdi)
268: {
269: String warFile = sdi.url.getFile();
270: return warFile.endsWith("war") || warFile.endsWith("war/");
271: }

396: public synchronized void start(DeploymentInfo di) throws DeploymentException
397: {
398: Thread thread = Thread.currentThread();
399: ClassLoader appClassLoader = thread.getContextClassLoader();
400: try
401: {
402: // Create a classloader for the war to ensure a unique ENC
403: URL[] empty = {};
404: URLClassLoader warLoader = URLClassLoader.newInstance(empty, di.ucl);
```

```
405: thread.setContextClassLoader(warLoader);
406: WebDescriptorParser webAppParser = new DescriptorParser(di);
407: String webContext = di.webContext;
408: if( webContext != null )
409: {
410:     if( webContext.length() > 0 && webContext.charAt(0) != '/' )
411:         webContext = "/" + webContext;
412: }
413: // Get the war URL
414: URL warURL = di.localUrl != null ? di.localUrl : di.url;
416: if (log.isDebugEnabled())
417: {
418:     log.debug("webContext: " + webContext);
419:     log.debug("warURL: " + warURL);
420:     log.debug("webAppParser: " + webAppParser);
421: }
423: // Parse the web.xml and jboss-web.xml descriptors
424: WebMetaData metaData = (WebMetaData) di.metaData;
425: parseMetaData(webContext, warURL, di.shortName, metaData);
426: WebApplication warInfo = new WebApplication(metaData);
427: warInfo.setDeploymentInfo(di);
428: performDeploy(warInfo, warURL.toString(), webAppParser);
429: deploymentMap.put(warURL.toString(), warInfo);
431: // Generate an event for the startup
432: super.start(di);
433: }
434: catch(DeploymentException e)
435: {
436:     throw e;
437: }
438: catch(Exception e)
439: {
440:     throw new DeploymentException("Error during deploy", e);
441: }
442: finally
443: {
444:     thread.setContextClassLoader(appClassLoader);
445: }
446: }

461: protected abstract void performDeploy(WebApplication webApp, String warUrl,
462: WebDescriptorParser webAppParser) throws Exception;

469: public synchronized void stop(DeploymentInfo di)
470: throws DeploymentException
471: {
```

```
472: URL warURL = di.localUrl != null ? di.localUrl : di.url;
473: String warUrl = warURL.toString();
474: try
475: {
476:     performUndeploy(warUrl);
477:     // Remove the web application ENC...
478:     deploymentMap.remove(warUrl);
479:     // Generate an event for the stop
480:     super.stop(di);
481: }
482: }
483: catch(DeploymentException e)
484: {
485:     throw e;
486: }
487: catch(Exception e)
488: {
489:     throw new DeploymentException("Error during deploy", e);
490: }
491: }

496: protected abstract void performUndeploy(String warUrl) throws Exception;

540: public void setConfig(Element config)
541: {
542: }

551: protected void parseWebAppDescriptors(DeploymentInfo di, ClassLoader loader,
552: WebMetaData metaData)
553: throws Exception
554: {
555:     log.debug("AbstractWebContainer.parseWebAppDescriptors, Begin");
556:     InitialContext iniCtx = new InitialContext();
557:     Context envCtx = null;
558:     Thread currentThread = Thread.currentThread();
559:     ClassLoader currentLoader = currentThread.getContextClassLoader();
560:     try
561:     {
562:         // Create a java:comp/env environment unique for the web application
563:         log.debug("Creating ENC using ClassLoader: "+loader);
564:         ClassLoader parent = loader.getParent();
565:         while( parent != null )
566:         {
567:             log.debug("..." +parent);
568:             parent = parent.getParent();
569:         }
570:         currentThread.setContextClassLoader(loader);
```

```
571: metaData.setENCLoader(loader);
572: envCtx = (Context) iniCtx.lookup("java:comp");
574: // Add a link to the global transaction manager
575: envCtx.bind("UserTransaction", new LinkRef("UserTransaction"));
576: log.debug("Linked java:comp/UserTransaction to JNDI name:
UserTransaction");
577: envCtx = envCtx.createSubcontext("env");
578: }
579: finally
580: {
581:   currentThread.setContextClassLoader(currentLoader);
582: }
584: Iterator envEntries = metaData.getEnvironmentEntries();
585: log.debug("addEnvEntries");
586: addEnvEntries(envEntries, envCtx);
587: Iterator resourceEnvRefs = metaData.getResourceEnvReferences();
588: log.debug("linkResourceEnvRefs");
589: linkResourceEnvRefs(resourceEnvRefs, envCtx);
590: Iterator resourceRefs = metaData.getResourceReferences();
591: log.debug("linkResourceRefs");
592: linkResourceRefs(resourceRefs, envCtx);
593: Iterator ejbRefs = metaData.getEjbReferences();
594: log.debug("linkEjbRefs");
595: linkEjbRefs(ejbRefs, envCtx, di);
596: Iterator ejbLocalRefs = metaData.getEjbLocalReferences();
597: log.debug("linkEjbLocalRefs");
598: linkEjbLocalRefs(ejbLocalRefs, envCtx, di);
599: String securityDomain = metaData.getSecurityDomain();
600: log.debug("linkSecurityDomain");
601: linkSecurityDomain(securityDomain, envCtx);
602: log.debug("AbstractWebContainer.parseWebAppDescriptors, End");
603: }

605: protected void addEnvEntries(Iterator envEntries, Context envCtx)
606:   throws ClassNotFoundException, NamingException
607: {
615: }

617: protected void linkResourceEnvRefs(Iterator resourceEnvRefs, Context envCtx)
618:   throws NamingException
619: {
649: }

651: protected void linkResourceRefs(Iterator resourceRefs, Context envCtx)
652:   throws NamingException
653: {
```



```
683: }

685: protected void linkEjbRefs(Iterator ejbRefs, Context envCtx, DeploymentInfo di)
686: throws NamingException
687: {
717: }

719: protected void linkEjbLocalRefs(Iterator ejbRefs, Context envCtx, DeploymentInfo di)
720: throws NamingException
721: {
763: }

774: protected void linkSecurityDomain(String securityDomain, Context envCtx)
775: throws NamingException
776: {
794: }

840: public String[] getCompileClasspath(ClassLoader loader)
841: {
864: }

1058: }
```

第 267~271 行，对应于由 JBoss 部署器实现的 `accepts` 方法，表明接受的部署单元类型。`AbstractWebContainer` 处理打包（jar 形式）或未打包（目录形式）的 WAR 部署单元。

第 396~446 行，对应于 `start` 方法，它是模板模式方法实现。`start` 方法参数是 `WAR DeploymentInfo` 对象。该对象含有 WAR 的 URL 和 `UnifiedClassLoader`、双亲存档（比如，EAR）以及 J2EE `application.xml` 的 `context-root`（如果 WAR 位于 EAR 中）。

`start` 方法首先保存当前线程上下文 `ClassLoader`，然后使用 WAR `UnifiedClassLoader` 创建另一 `URLClassLoader`（`warLoader`），并将它作为 WAR 的双亲，即当前线程上下文 `ClassLoader`。其中，创建的 `warLoader` 用于确保该 WAR JNDI ENC 的惟一性（对应于第 403~404 行）。第 3 章提到，创建 `java:comp` 上下文的 `ClassLoader` 决定了 `java:comp` 上下文的惟一性。第 405 行，即调用 `performDeploy` 之前，代码将 `warLoader` 设置为当前线程上下文 `ClassLoader`。接下来，第 425 行，通过调用 `parseMetaData` 方法分析 `web.xml` 和 `jboss-web.xml` 描述符。再然后，第 428 行，`performDeploy` 方法要求 Web 容器的自身子类完成 WAR 的实际部署。第 429 行，将 `WebApplication` 对象存储在 `deploymentMap` 中，并使用 `warUrl` 作为其键（key）。最后一步，第 444 行，恢复先前保存的线程上下文 `ClassLoader`。

第 461~462 行给出了抽象 `performDeploy` 方法的方法名和定义。该方法供 `start` 方法调用，并且完成 Web 应用实际部署任务的子类必须覆盖它。其中，将 `WebApplication` 作为 `performDeploy` 方法的参数，它含有 `web.xml` 和 `jboss-web.xml` 描述符的元数据。这些元数据含有 J2EE `application.xml` 描述符中 Web 模块的 `context-root` 值，或者是单独 Web 部署单元中 `jboss-web.xml` 描述符的 `context-root` 值。另外，这些元数据还包含 `jboss-web.xml` 描述符中可能存在的 `virtual-host` 值。一旦从 `performDeploy` 方法返回，必须使用 `Servlet` 上下

文的 `ClassLoader` 部署 `WebApplication`。其中, `warUrl` 参数是待部署 Web 应用 URL 的字符串表示。`webAppParser` 参数是回调 `handle`, 即子类必须使用它调用 `parseWebAppDescriptors` 方法, 才能建立 Web 应用的 JNDI 环境。该回调为子类提供了钩子 (hook), 它能保证 WAR 启动时装载的任何 Servlet 只有在建立 Web 应用的 JNDI 环境后才会被创建。另外, 子类必须正确实现 `performDeploy` 方法, 即在启动需要访问 JBoss JNDI (比如 EJB 和资源工厂等) 的 Servlet 之前, 能够调用 `parseWebAppDescriptors` 方法。最后, 在实现子类过程中, 值得注意的一个重要设置细节是, 它必须将当前线程上下文 `ClassLoader` 作为具体 Web 容器创建的类装载器的双亲 `ClassLoader`。否则, Web 应用通过 JNDI ENC 访问 EJB 或 JBoss 资源会出现很多问题。

第 469~491 行, 对应于 `stop` 方法, 它是模板模式方法实现。第 476 行, 调用了 `performUndeploy` 方法, 供具体容器完成卸载任务使用。接下来, 第 478 行, 将注册的 `warUrl` 从 `deploymentMap` 删除。字符串形式的 `warUrl` 参数源于先前传入到 `performDeploy` 方法中的 WAR URL。

第 496 行, 给出了抽象 `performUndeploy` 方法的方法名和定义。位于第 476 行的 `stop` 方法模板调用了该方法。`performUndeploy` 要求子类完成具体 Web 容器的相关卸载任务。

第 540~542 行, 对应于 `setConfig` 方法。如果通过 MBean 属性不能够满足子类的扩展要求, 则子类可以重载该方法。`config` 参数是含有特定层次结构的双亲 DOM 元素, 它来自 Web 容器服务 MBean `jboss-service.xml` 描述符中 `Config` 属性的子元素。本文后续研究支持将嵌入式 Tomcat 集成到 JBoss 的 MBean 时, 开发者能够体会到 `setConfig` 方法使用和 `config` 值。

第 551~603 行, 对应于 `parseWebAppDescriptors` 方法。当子类 `performDeploy` 方法在调用 `webAppParser.parseWebAppDescriptors` 回调时, 该方法将会设置声明在 `web.xml` 描述符中的 Web 应用 ENC (`java:comp/env`) `env-entry`、`resource-env-ref`、`resource-ref`、`local-ejb-ref`、`ejb-ref` 元素值。其中, 创建 `env-entry` 值不需要 `jboss-web.xml` 描述符; 创建 `resource-env-ref`、`resource-ref` 以及 `ejb-ref` 元素需要使用 `jboss-web.xml` 描述符中已部署资源和 EJB 的 JNDI 名。由于 ENC 上下文是 Web 应用的私有内容, 所以 Web 应用 `ClassLoader` 惟一标识了 ENC。`loader` 参数代表 Web 应用的 `ClassLoader`, 并且不能为 `null`。`metaData` 参数是传入子类 `performDeploy` 方法的 `WebMetaData` 参数。第 584~601 行, 给出了 `parseWebAppDescriptors` 方法是如何使用 WAR 部署描述符中的元数据信息, 以及调用其他方法创建 JNDI ENC 绑定过程的。

第 605~615 行, 对应于 `addEnvEntries` 方法。它创建了 `web.xml` 描述符指定的 `java:comp/env` Web 应用 `env-entry` 绑定。

第 617~649 行, 对应于 `linkResourceEnvRefs` 方法。它将 `web.xml` 描述符 `resource-env-ref` 元素指定的 `java:comp/env/xxx` Web 应用 JNDI ENC 映射到 `jboss-web.xml` 描述符指定的已部署 JNDI 名上。

第 653~683 行, 对应于 `linkResourceRefs` 方法。它将 `web.xml` 描述符 `resource-ref` 元素指定的 `java:comp/env/xxx` Web 应用 JNDI ENC 映射到 `jboss-web.xml` 描述符指定的已部署 JNDI 名上。

第 685~717 行, 对应于 `linkEjbRefs` 方法。它将 `web.xml` 描述符 `ejb-ref` 元素指定的

java:comp/env/ejb Web 应用 JNDI ENC 映射到 jboss-web.xml 描述符指定的已部署 JNDI 名上。

第 719~763 行，对应于 linkEjbLocalRefs 方法。它将 web.xml 描述符 ejb-local-ref 元素指定的 java:comp/env/ejb Web 应用 JNDI ENC 映射到 jboss-web.xml 描述符中用 ejb-link 映射指定的已部署 JNDI 名上。

第 774~794 行，对应于 linkSecurityDomain 方法，它创建 java:comp/env/security 上下文。该上下文包含指向 AuthenticationManager 实现的 securityMgr 绑定和指向 RealmMapping 实现的 realmMapping 绑定。其中，realmMapping 实现和 Web 应用的安全性域关联。另外，还将创建 Subject 绑定，供动态访问请求线程的已认证 Subject 使用。如果 jboss-web.xml 描述符含有 security-domain 元素，则 javax.naming.LinkRefs 绑定到 security-domain 元素指定的 JNDI 名上，或者 JNDI 名的子上下文。如果描述符中不存在 security-domain 元素，则绑定到 org.jboss.security.plugins.NullSecurityManager 实例，即简单地实现所有认证和授权检查。

第 840~864 行，对应于 getCompileClasspath 方法。它是为 Web 容器提供的实用方法，即从给定类装载器开始浏览 ClassLoader 链，并查询各个 ClassLoader 以创建 URL 字符串形式的完整类路径。某些 JSP 编译器实现（比如，Jasper）需要该方法，供获得编译所需的完整类路径。

9.1.2 创建 AbstractWebContainer 子类

为将目标 Web 容器集成到 JBoss 中，开发者需要实现 AbstractWebContainer 子类，并按要求实现上述描述的 performDeploy(WebApplication,String,WebDescriptorParser) 和 performUndeploy(String) 方法。另外，在集成过程中，需要注意如下事项。

1. 使用线程上下文类装载器

尽管在上述 performDeploy 方法描述中已经提到这个问题，但由于这个问题很重要，因此这里再次强调。在设置 WAR 容器期间，子类实现必须将当前线程上下文 ClassLoader 作为具体 Web 容器创建的类装载器的双亲 ClassLoader。否则，Web 应用通过 JNDI ENC 访问 EJB 或 JBoss 资源会出现很多问题。

2. 使用 Log4j 集成日志功能

JBoss 使用 Apache Log4j 作为其内部日志功能 API。为很好地实现 Web 容器同 JBoss 的集成，JBoss 需要提供 Web 容器日志抽象到 Log4j API 的映射。AbstractWebContainer 子类需要借助于超类的 log 实例变量或 getLog 方法访问 Log4j 接口。该接口包裹了 Log4j category 的 org.jboss.logging.Logger 类。其中，这里的 Log4j category 名是容器子类名。

3. 将 Web 容器认证和授权委派给 JBossSX

理想状态下，Web 应用和 EJB 的认证和授权应该由同一安全性管理器处理。为实现这种理想状态，Web 容器必须能够钩到 JBoss 安全性层。通常，这要求存在请求拦截器，将 Web 容器安全性请求传回 JBoss 安全性 API 处理。其中，同 JBossSX 安全性框架集成的前提是，建立了前面讨论 linkSecurityDomain 方法时提到的“java:comp/env/security”上下文。该安全性上下文为访问 JBossSX 安全性管理器接口实现提供支持，从而子类的请求

拦截器能够为 Web 应用提供安全性服务。列表 9-3 给出了使用安全性上下文认证用户的大体步骤（用伪码描述）。列表 9-4 给出了使用安全性上下文授权用户的大体步骤（用伪码描述）。

列表 9-3 借助于 JBossSX API 和 java:comp/env/security JNDI 上下文认证用户（用伪码描述）

```
// Get the username and password from the request context...
HttpServletRequest request = ...;
String username = getUsername(request);
String password = getPassword(request);
// Get the JBoss security manager from the ENC context
InitialContext iniCtx = new InitialContext();
AuthenticationManager securityMgr = (AuthenticationManager)
iniCtx.lookup("java:comp/env/security/securityMgr");
SimplePrincipal principal = new SimplePrincipal(username);
if( securityMgr.isValid(principal, password) )
{
    // Indicate the user is allowed access to the web content...
    // Propagate the user info to JBoss for any calls into made by the servlet
    SecurityAssociation.setPrincipal(principal);
    SecurityAssociation.setCredential(password.toCharArray());
}
else
{
    // Deny access...
}
```

列表 9-4 借助于 JBossSX API 和 java:comp/env/security JNDI 上下文授权用户（用伪码描述）

```
// Get the username & required roles from the request context...
HttpServletRequest request = ...;
String username = getUsername(request);
String[] roles = getContentRoles(request);
// Get the JBoss security manager from the ENC context
InitialContext iniCtx = new InitialContext();
RealmMapping securityMgr = (RealmMapping)
iniCtx.lookup("java:comp/env/security/realmMapping");
SimplePrincipal principal = new SimplePrincipal(username);
Set requiredRoles = new HashSet(java.util.Arrays.asList(roles));
if( securityMgr.isUserHaveRole(principal, requiredRoles) )
{
    // Indicate user has the required roles for the web content...
}
else
{
    // Deny access...
}
```


9.2 JBoss/Tomcat-4.1.x 绑定

本节内容将讨论 JBoss/Tomcat-4.x 集成绑定中的具体配置问题。Tomcat-4.1.x 发布版，即著名的 Catalina，是最新的 Apache Java Servlet 容器。它支持 Servlet 2.3 和 JSP 1.2 规范。JBoss MBean 服务配置控制了 JBoss/Tomcat 集成层。用于嵌入 Tomcat-4.1.x 系列容器的服务是 `org.jboss.web.tomcat.tc4.EmbeddedTomcatService` MBean，它是 `AbstractWebContainer` 的子类。其可配置属性如下：

- **CatalinaHome:** 设置 `catalina.home` 系统属性值，用于指定 JBoss 结构外部的 Catalina 发布版位置。如果没有指定 `CatalinaHome` 属性，则根据包含 `org.apache.catalina.startup.Embedded` 类的 jar 位置给出 `catalina.home` 系统属性值（假定是标准 Catalina 发布版结构）。
- **CatalinaBase:** 设置 `catalina.base` 系统属性值，用于解析相对路径。如果没有指定，则使用 `CatalinaHome` 属性值。
- **Java2ClassLoadingCompliance:** 生效标准 Java 2 双亲委派类装载模型，而不是使用 Servlet 2.3 类装载模型。默认值为 `true`，即使用 Java 2 双亲委派类装载模型，因此从 WAR 装载了 EJB 使用的客户端 jar 时会触发类装载冲突。如果改为 `false`，即生效 Servlet 2.3 类装载模型，则需要整理部署包，以避免部署单元中存在重复类。
- **DeleteWorkDirs:** 标志位，表明出现卸载事件时是否删除工作目录。默认时，即停止某上下文时，Catalina 不会删除其工作目录。因此，重新部署 JSP 页面时便不需要重新编译 JSP，除非 WAR 中文件的时间戳已经更新。默认值为 `true`。
- **SnapshotMode:** 设置群集环境中的快照模式，其值或者为“`instant`”，或者为“`interval`”。在 `instant` 模式时，会话一旦发生修改，则会立即传播群集会话变化。如果是 `interval` 模式，则基于 `SnapshotInterval` 属性收集所有的修改，并定期传播。
- **SnapshotInterval:** 为 `interval` 快照模式设置快照间隔（单位：毫秒）。默认值为 1000，即 1 秒。
- **Config:** 提供扩展配置的属性，即使用标准 Tomcat `server.xml` 文件指定其他连接器等内容的构造块。请注意，这只适合于嵌入式 Tomcat Servlet 容器，即没有使用 Tomcat 配置文件，比如 `conf/server.xml`。图 9-2 给出了当前支持的配置 DTD 大纲，下一节内容将解释图中的元素。
- **UseJBossWebLoader:** 标志位，表明 Tomcat 用于 Web 应用类装载器的类装载器是否是 JBoss 统一类装载器。默认值为 `true`，即表示将 WAR 中 `WEB-INF/classes` 和 `WEB-INF/lib` 中的可用类集成到默认共享类装载器库中（第 2 章已阐述）。同 Servlet 2.3 类装载模型相比，这可能不能满足开发者的要求，因为它会导致类和资源在不同 Web 应用中共享。将该属性设置为 `false`，则不使用 JBoss 统一类装载器。

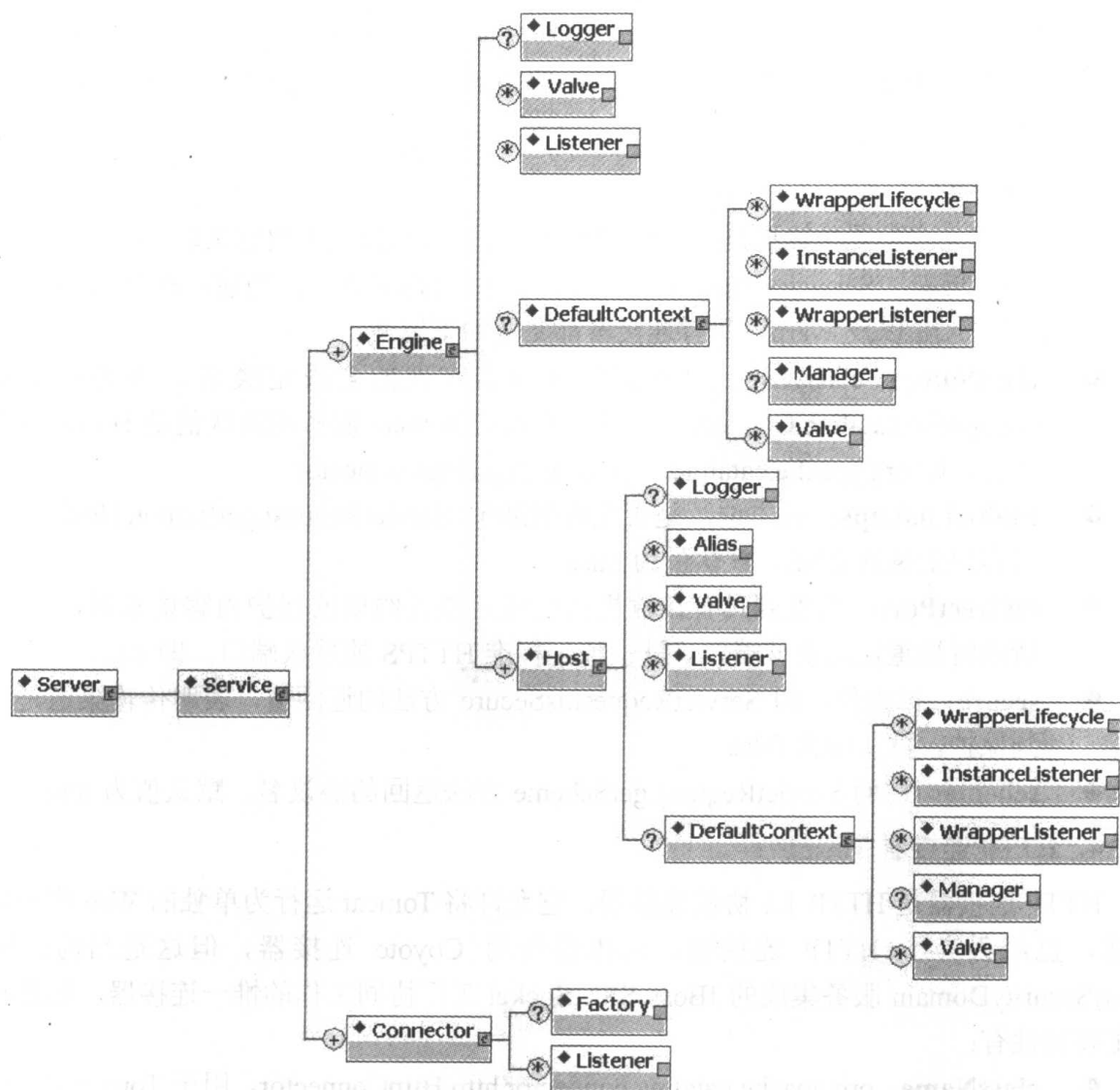


图 9-2 Tomcat-4.1.24 配置 DTD 概貌 (支持 EmbeddedCatalinaService41 Config 属性)

集成 Tomcat 和 JBoss 需要借助于 jboss-service.xml 描述符。其中, 该描述符位于 default 和 all 配置文件集合的 deploy/jbossweb-tomcat.sar/META-INF 目录下。在 JBoss/Tomcat 绑定发布版中, 只能找到这个描述符。另外, 该绑定还将包含的 Tomcat 发布版放置在 jbossweb-tomcat41.sar 目录下。当前, JBoss 3.2.3 绑定的 Tomcat 发布版版本是 jakarta-tomcat-4.1.29-LE-jdk14。

9.2.1 嵌入式 Tomcat 配置元素

本节介绍 Tomcat 配置元素, 这些元素有可能是 EmbeddedTomcatService Config 属性的子元素。

1. Server

Server 元素是 Tomcat Servlet 容器配置的根本元素。目前, 嵌入式服务不支持 Server 元素属性。

2. Service

Service 是一个或多个连接器，以及单个引擎的容器。其仅支持的属性有：name，服务的惟一名。

3. Connector

Connector 元素配置传输机制，即允许客户发送请求和接收来自它关联 Service 的响应。连接器将请求传递给 Service Engine，并将结果返回给请求客户。当前，存在 3 个连接器实现：HTTP、AJP 以及 Warp。所有连接器都支持如下属性：

- **className**：className 属性指定连接器实现的全限定类名。该类必须实现 org.apache.catalina.Connector 接口。嵌入式 Tomcat 服务的默认值是 HTTP 连接器实现，即 org.apache.catalina.connector.http.HttpConnector。
- **enableLookups**：标志位，是否允许借助于 ServletRequest.getRemoteHost 方法解析客户主机的 DNS。默认值为 false。
- **redirectPort**：当收到具有传输机密性或完整性约束的保护内容请求时，非 SSL 请求将被重定向到该端口。默认值为标准 HTTPS 的默认端口，即 443。
- **secure**：标志位，即 ServletRequest.isSecure 方法的返回值，表明传输渠道是否受到保护。默认值为 false。
- **scheme**：访问 ServletRequest.getScheme 方法返回的协议名。默认值为 http。

4. HTTP 连接器

HTTP 连接器是 HTTP 1.1 协议连接器，它允许将 Tomcat 运行为单独的 Web 服务器。目前，已经丢弃了 HTTP 连接器，而推荐使用 Coyote 连接器，但这是当前能够同 JBossSecurityDomain 服务集成的 JBoss SSL Socket 工厂协同工作的惟一连接器。该连接器的主要属性有：

- **className**：org.apache.catalina.connector.http.HttpConnector，用于 Tomcat-4.1.x 版本的 Web 容器。
- **acceptCount**：当所有可能请求处理线程都占用时，所允许的连接请求的最大队列长度。当队列已满时，任何收到的请求都将被拒。默认值为 10。
- **allowChunking**：如果设置为 true，则当处理 HTTP/1.1 请求时，允许分块(chunked)输出。默认值为 true。
- **address**：对于含有多个 IP 地址的服务器而言，address 属性值指定使用哪个地址监听指定的端口。默认时，端口作用于服务器的所有 IP 地址。
- **bufferSize**：连接器创建的输入流缓存区大小（单位：字节）。默认值为 2 048 字节。
- **connectionTimeout**：连接器接收到连接后，等待请求的时间（单位：毫秒）。默认值 60 000 毫秒，即 60 秒。
- **debug**：连接器生成日志消息的调试详细程度，值越大，输出越详细。如果没有指定，则为 0。无论是否指定 debug 属性，输出信息的详细程度最终取决于 Log4j category “org.jboss.web.catalina.EmbeddedCatalinaService41”。
- **maxProcessors**：连接器创建的最大请求处理线程数量。它决定了能够处理的最大并发请求数量。默认值为 20。

- **minProcessors**: 连接器初次启动时, 创建的最小请求处理线程数量。minProcessors 值应该小于 maxProcessors 值。默认值为 5。
- **port**: 连接器在 port 属性值上创建服务器 Socket 的 TCP 端口号, 并等待连接。操作系统有如下限定, 即处于特定 IP 地址上的服务器应用同时只能监听单个端口。
- **proxyName**: 如果连接器用于代理配置中, 则将 proxyName 属性设置成 request.getServerName()调用返回的服务器名。更多信息, 请参考代理支持¹。
- **proxyPort**: 如果连接器用于代理配置中, 则将 proxyPort 属性设置成 request.getServerPort()调用返回的服务器端口。更多信息, 请参考代理支持。
- **tcpNoDelay**: 如果设置为 true, 则服务器 Socket 将设置 TCP_NO_DELAY 选项, 这在大部分场合都能提高性能。默认值为 true。

HTTP 连接器的其他属性请参考 Tomcat 网站文档:

<http://jakarta.apache.org/tomcat/tomcat-4.1-doc/config/http11.html>。

5. Coyote 连接器

CoyoteConnector 是默认 HTTP/1.1 连接器。借助于该连接器, Tomcat 除了能执行 Servlet 和 JSP 页面外, 还能够以单独的 Web 服务器方式运行。另外, 还可以配置 Apache mod_jk 和 mod_jk2 模块, 以处理 AJP 1.3 协议。Coyote 连接器支持的属性有:

- **className**: org.apache.coyote.tomcat4.CoyoteConnector, 用于 Tomcat-4.1.x 版本的 Web 服务器。
- **acceptCount**: 当所有可能请求处理线程都占用时, 所允许的连接请求的最大队列长度。当队列已满时, 任何收到的请求都将被拒。默认值为 10。
- **address**: 对于含有多个 IP 地址的服务器而言, address 属性值指定使用哪个地址监听指定的端口。默认时, 端口作用于服务器的所有 IP 地址。
- **bufferSize**: 连接器创建的输入流缓存区大小 (单位: 字节)。默认值为 2 048 字节。
- **compression**: 连接器可能为节省服务器带宽, 使用了 HTTP/1.1 GZIP 压缩。其取值范围如下: off (不启用压缩)、on (启动压缩, 但只压缩文本数据)、force (无论何时, 都强制压缩) 或者整数 (等价于 on, 但是压缩输出数据前指定了最小数据量)。如果 content-length 不详, 并且 compression 设置为 on 或压缩程度更高的取值, 也将压缩输出。默认值为 off。
- **connectionLinger**: 当连接器使用的 Socket 关闭时, 保持 Socket 的时间 (单位: 毫秒)。默认值为 -1 (不使用 Socket Linger)。
- **connectionTimeout**: 连接器接收到连接后, 等待请求的时间 (单位: 毫秒)。默认值为 60 000 毫秒, 即 60 秒。
- **debug**: 连接器生成日志消息的调试详细程度, 值越大, 输出越详细。如果没有指定, 则为 0。无论是否指定 debug 属性, 输出信息的详细程度最终取决于 Log4j category “org.jboss.web.catalina.EmbeddedCatalinaService41”。
- **disableUploadTimeout**: 标志位。它表明, 当在执行 Servlet 时, 是否允许 Servlet

¹ 译者注: 请参考 Tomcat 网站在线文档, “Proxy Support” 节内容。

容器使用不同的、更长的连接超时。使用该标志位的结果是，或者 Servlet 完成执行消耗了更长的超时时间，或者数据上传期间出现了更长的超时。默认值为 false。

- **maxProcessors**: 连接器创建的最大请求处理线程数量。它决定了能够处理的最大并发请求数量。默认值为 20。
- **minProcessors**: 连接器初次启动时，创建的最小请求处理线程数量。minProcessors 值应该小于 maxProcessors 值。默认值为 5。
- **port**: 连接器在 port 属性值上创建服务器 Socket 的 TCP 端口号，并等待连接。操作系统有如下限定，即处于特定 IP 地址上的服务器应用只能够监听一个端口。
- **proxyName**: 如果连接器用于代理配置中，则将 proxyName 属性设置成 request.getServerName()调用返回的服务器名。更多信息，请参考代理支持。
- **proxyPort**: 如果连接器用于代理配置中，则将 proxyPort 属性设置成 request.getServerPort()调用返回的服务器端口。更多信息，请参考代理支持。
- **tcpNoDelay**: 如果设置为 true，则服务器 Socket 将设置 TCP_NO_DELAY 选项，这在大部分场合都能提高性能。默认值为 true。

Coyote 连接器的其他属性请参考 Tomcat 网站页面：

<http://jakarta.apache.org/tomcat/tomcat-4.1-doc/config/coyote.html>。

6. Engine

每个 Service 必须提供单个 Engine 配置。Engine 处理借助于配置的连接器的提交给 Service 的请求。嵌入式 Tomcat 服务支持的 Engine 子元素包括：Host、Logger、DefaultContext、Valve 以及 Listener。Engine 支持的属性有：

- **className**: org.apache.catalina.Engine 接口实现的全限定类名。默认值为 org.apache.catalina.core.StandardEngine。
- **defaultHost**: 配置在 Engine 下的 Host 名，用于处理主机名不匹配 Host 配置请求。
- **name**: 分配给 Engine 的逻辑名。Engine 生成日志消息时会使用到 name 属性。

Engine 元素的其他信息请参考 Tomcat 网站文档：

<http://jakarta.apache.org/tomcat/tomcat-4.1-doc/config/engine.html>。

7. Host

Host 元素给出虚拟主机配置。它是带有指定 DNS 主机名的，并且为 Web 应用提供的 Web 容器。嵌入式 Tomcat 服务支持的 Host 子元素有：Alias、Logger、DefaultContext、Valve 以及 Listener。Host 支持的属性有：

- **className**: 指定 org.apache.catalina.Host 接口实现的全限定类名。默认值为 org.apache.catalina.core.StandardHost。
- **name**: 虚拟主机的 DNS 名。Host 元素至少必须配置 name 属性，并且能够对应于含有 Host 元素的 Engine 的 defaultHost 值。

Host 元素的其他信息请参考 Tomcat 网站文档：

<http://jakarta.apache.org/tomcat/tomcat-4.1-doc/config/host.html>。

8. Alias

Alias 元素是 Host 元素的可选子元素。每个 Alias 内容指定相应 Host 的另一 DNS 名。

9. DefaultContext

DefaultContext 元素是用于 Web 应用上下文的配置模板，可以在 Engine 或 Host 级别定义 DefaultContext 元素。嵌入式 Tomcat 服务支持的 DefaultContext 子元素有：WrapperLifecycle、InstanceListener、WrapperListener 以及 Manager。DefaultContext 支持的属性有：

- **className**: org.apache.catalina.core.DefaultContext 实现的全限定类名。默认值为 org.apache.catalina.core.DefaultContext。如果不使用默认值，则提供的类必须是 DefaultContext 的子类。
- **cookies**: 标志位，表明是否使用 Cookie 跟踪会话。默认值为 true。
- **crossContext**: 标志位，表明 ServletContext.getContext(String path)方法是否为部署在调用 Web 应用的虚拟主机上的其他 Web 应用返回上下文。默认值为 false。

10. Manager

Manager 元素定义了会话管理器，它是 DefaultContext 配置的可选子元素。Manager 元素支持的属性有：

- **className**: org.apache.catalina.Manager 接口实现的全限定类名。默认值为 org.apache.catalina.session.StandardManager。

11. Logger

Logger 元素为 Engine、Host 以及 DefaultContext 指定日志功能配置。其支持的属性有：

- **className**: org.apache.catalina.Logger 接口实现的全限定类名。默认值为 org.jboss.web.catalina.Log4jLogger。应该使用 className 同 JBoss 服务器 Log4j 系统集成。

12. Valve

Valve 元素配置了请求管道 (pipeline) 元素。其中，Valve 元素值实现了 org.apache.catalina.Valve 接口，存在若干个标准 Valve 实现供使用。用户经常使用 Valve 记录访问请求。Valve 元素支持的属性有：

- **className**: org.apache.catalina.Valve 接口实现的全限定类名。其取值必须是 org.jboss.web.catalina.valves.AccessLogValue。
- **directory**: 访问日志文件保存的目录路径。
- **pattern**: 定义日志信息的格式。默认值为 common。
- **prefix**: 添加到各个日志文件名的前缀。默认值为 “access_log”。
- **suffix**: 添加到各个日志文件名的后缀。默认值为 “”，即不添加任何后缀。

Valve 元素的其他信息和可用的 Valve 实现，请参考 Tomcat 网站文档：

<http://jakarta.apache.org/tomcat/tomcat-4.1-doc/config/valve.html>。

13. Listener

Listener 元素配置组件生命周期监听器。为添加生命周期监听器，用户需要使用 className 属性指定 org.apache.catalina.LifecycleListener 接口实现的全限定类名，以及该监听器实现支持的其他属性。

9.2.2 JBoss/Tomcat 绑定使用 SSL

存在若干种方式为嵌入式 Tomcat Servlet 容器配置使用基于 SSL 的 HTTP。这些不同方式间的主要差别在于，是否使用 JBoss 提供的连接器 Socket 工厂，即允许从 JBossSX SecurityDomain 获得 JSSE 服务器证书信息。但前提是，要求使用 org.jboss.security.plugins.JaasSecurityDomain MBean 创建 SecurityDomain。上述两个过程同第 8 章研究的基于 SSL 加密使用 RMI 很类似。列表 9-5 给出借助于该办法，并且仅设置 SSL 连接器时的 jbossweb-tomcat41.sar/META-INF/jboss-service.xml 配置文件。该配置包含了第 8 章的同一 JaasSecurityDomain 设置，但由于这里的描述符没有部署在包含 chap8.keystore 文件的 SAR 中，因此需要将 chap8.keystore 拷贝到 server/default/conf 目录中。

列表 9-5 JaasSecurityDomain 和 EmbeddedCatalinaSX MBean 配置
(设置 Tomcat-4.x 使用 SSL 作为首选连接器协议)

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- An example tomcat config that only uses SSL connectors.
-->

<!-- The service configuration for the embedded Tomcat4 web container
-->
<server>

  <!-- The SSL domain setup -->
  <mbean code="org.jboss.security.plugins.JaasSecurityDomain"
    name="Security:name=JaasSecurityDomain,domain=RMI+SSL">
    <constructor>
      <arg type="java.lang.String" value="RMI+SSL"/>
    </constructor>
    <attribute name="KeyStoreURL">chap8.keystore</attribute>
    <attribute name="KeyStorePass">.rmi+ssl</attribute>
  </mbean>

  <mbean code="org.jboss.web.tomcat.tc4.EmbeddedTomcatService"
    name="jboss.web:service=WebServer">

    <!-- Get the flag indicating if the normal Java2 parent first class
    loading model should be used over the servlet 2.3 web container first
    model.
    -->
    <attribute name="Java2ClassLoadingCompliance">true</attribute>
```

```

<attribute name="LenientEjbLink">true</attribute>

<!-- A flag indicating if the JBoss Loader should be used. This loader
uses a unified class loader as the class loader rather than the tomcat
specific class loader.
-->
<attribute name="UseJBossWebLoader">true</attribute>

<!-- The name of the request attribute under with the authenticated JAAS
Subject is stored on successful authentication. If null or empty then
the Subject will not be stored.
-->
<attribute name="SubjectAttributeName">j_subject</attribute>

<attribute name="Config">
  <Server>
    <Service name="JBoss-Tomcat">
      <Engine name="MainEngine" defaultHost="localhost">
        <Logger className="org.jboss.web.tomcat.Log4jLogger"
          verbosityLevel="debug" category="org.jboss.web.localhost.Engine"/>
        <Host name="localhost">

          <!-- Access logger -->
          <Valve className="org.apache.catalina.valves.AccessLogValve"
            prefix="localhost_access" suffix=".log"
            pattern="common" directory="{jboss.server.home.dir}/log"/>

          <!-- This valve clears any caller identity set by the realm
and provides access to the realm about the existence of an
authenticated caller to allow a web app to run with a realm
that support unauthenticated identities. It also establishes
any run-as principal for the servlet being accessed.
-->
          <Valve className="org.jboss.web.tomcat.security.SecurityAssociationValve"/>
          <!-- Default context parameters -->
          <DefaultContext cookies="true" crossContext="true" override="true"/>
        </Host>
      </Engine>

      <!-- SSL/TLS Connector configuration using the SSL domain keystore -->
      <Connector className = "org.apache.catalina.connector.http.HttpConnector"
        port = "443" scheme = "https" secure = "true">
        <Factory className =
          "org.jboss.web.tomcat.security.SSLServerSocketFactory"
          securityDomainName = "java:/jaas/RMI+SSL" clientAuth = "false"
          protocol = "TLS"/>
      </Connector>
    </Service>
  </Server>
</attribute>

```



```

        </Service>
    </Server>
</attribute>
</mbean>

</server>

```

使用如下 URL: <https://localhost/jmx-console/index.jsp>, 能够访问到 JMX 控制台 Web 应用, 即测试上述配置。



如果目标操作系统是 *nix (Linux, Solaris 以及 OS X), 则只有具有 root 身份的用户才能够打开小于 1024 的端口。因此, 需要将上述端口改为大于 1024 的端口, 比如 8443。

或者, 如果开发者想同时支持非 SSL 和 SSL 方式访问, 则需要为 Embedded CatalinaService41 MBean 添加其他连接器配置。列表 9-6 给出的 jboss-service.xml 配置文件实例, 它演示了满足该 SSL 条件的配置。

列表 9-6 JaasSecurityDomain 和 EmbeddedCatalinaSX MBean 配置 (设置 Tomcat-4.x 使用非 SSL 和 SSL 作为 HTTP 连接器)

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- An example tomcat config that only uses both non-SSL and SSL connectors.
-->

<!-- The service configuration for the embedded Tomcat4 web container
-->
<server>

    <!-- The SSL domain setup -->
    <mbean code="org.jboss.security.plugins.JaasSecurityDomain"
name="Security:name=JaasSecurityDomain,domain=RMI+SSL">
        <constructor>
            <arg type="java.lang.String" value="RMI+SSL"/>
        </constructor>
        <attribute name="KeyStoreURL">chap8.keystore</attribute>
        <attribute name="KeyStorePass">.rmi+ssl</attribute>
    </mbean>

    <mbean code="org.jboss.web.tomcat.tc4.EmbeddedTomcatService"
name="jboss.web:service=WebServer">

        <!-- Get the flag indicating if the normal Java2 parent first class
loading model should be used over the servlet 2.3 web container first
model.
-->

```

```

<attribute name="Java2ClassLoadingCompliance">true</attribute>

<attribute name="LenientEjbLink">true</attribute>

<!-- A flag indicating if the JBoss Loader should be used. This loader
uses a unified class loader as the class loader rather than the tomcat
specific class loader.
-->
<attribute name="UseJBossWebLoader">true</attribute>

<!-- The name of the request attribute under with the authenticated JAAS
Subject is stored on successful authentication. If null or empty then
the Subject will not be stored.
-->
<attribute name="SubjectAttributeName">j_subject</attribute>

<attribute name="Config">
  <Server>
    <Service name="JBoss-Tomcat">
      <Engine name="MainEngine" defaultHost="localhost">
        <Logger className="org.jboss.web.tomcat.Log4jLogger"
        verbosityLevel="debug" category="org.jboss.web.localhost.Engine"/>
        <Host name="localhost">

          <!-- Access logger -->
          <Valve className="org.apache.catalina.valves.AccessLogValve"
          prefix="localhost_access" suffix=".log"
          pattern="common" directory="{jboss.server.home.dir}/log"/>

          <!-- This valve clears any caller identity set by the realm
and provides access to the realm about the existence of an
authenticated caller to allow a web app to run with a realm
that support unauthenticated identities. It also establishes
any run-as principal for the servlet being accessed.
-->
          <Valve className="org.jboss.web.tomcat.security.SecurityAssociationValve"/>
          <!-- Default context parameters -->
          <DefaultContext cookies="true" crossContext="true" override="true"/>
        </Host>
      </Engine>

      <!-- HTTP Connector configuration -->
      <Connector className = "org.apache.catalina.connector.http.HttpConnector"
      port = "8080" redirectPort = "443"/>
      <!-- SSL/TLS Connector configuration using the SSL domain keystore -->

```

```
<Connector className = "org.apache.catalina.connector.http.HttpConnector"
port = "443" scheme = "https" secure = "true">
  <Factory className =
    "org.jboss.web.catalina.security.SSLServerSocketFactory"
    securityDomainName = "java:/jaas/RMI+SSL" clientAuth = "false"
    protocol = "TLS"/>
</Connector>
</Service>
</Server>
</attribute>
</mbean>

</server>
```

同时，开发者也可以使用标准 Coyote 连接器配置 SSL。但是，Coyote 连接器并不能很好地支持自定义 SSL Socket 工厂，而且还不能够使用 `org.jboss.web.catalina.security.SSLServerSocketFactory`。如果使用 SSL，则需要为 Coyote 连接器添加 Factory 元素，以配置要求的 SSL Socket 工厂。Factory 元素支持如下属性：

- **algorithm**：使用的证书编码算法。默认值为 `SunX509`。
- **className**：SSL 服务器 Socket 工厂实现类的全限定类名。其中，className 的取值必须是 `org.apache.coyote.tomcat4.CoyoteServerSocketFactory`。使用其他类型的 Socket 工厂并不会出现错误，但 `CoyoteServerSocketFactory` 创建的服务器 Socket 不支持 SSL 的使用。
- **clientAuth**：如果设置为 `true`，则 SSL 栈希望在接受客户连接前，能够收到有效的证书链。默认值为 `false`，即不要求证书链，除非客户请求的资源使用了 CLIENT-CERT 认证安全性约束保护。
- **keystoreFile**：keystore 文件的路径名，其中存储了待装载的服务器证书。默认时，该路径名是操作系统中运行 Tomcat 用户的主（home）目录中的“.keystore”文件。
- **keystorePass**：访问指定 keystore 文件中服务器证书的密码。默认值为“changeit”。
- **keystoreType**：用于存储服务器证书的 keystore 文件类型。默认值为“JKS”。
- **protocol**：SSL 协议版本。默认值为“TLS”。

列表 9-7 给出了满足上述 SSL 配置条件的 jboss-service.xml 文件。

列表 9-7 EmbeddedCatalinaSX MBean 配置（设置 Tomcat-4.x 使用 SSL 作为 Coyote 连接器协议）

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- An embedded Tomcat4.1.x web container that uses only an SSL/Coyote connector -->
<server>

  <mbean code="org.jboss.web.tomcat.tc4.EmbeddedTomcatService"
    name="jboss.web:service=WebServer">
```

```

<!-- Get the flag indicating if the normal Java2 parent first class
loading model should be used over the servlet 2.3 web container first
model.
-->
<attribute name="Java2ClassLoadingCompliance">true</attribute>

<attribute name="LenientEjbLink">true</attribute>

<!-- A flag indicating if the JBoss Loader should be used. This loader
uses a unified class loader as the class loader rather than the tomcat
specific class loader.
-->
<attribute name="UseJBossWebLoader">true</attribute>

<!-- The name of the request attribute under with the authenticated JAAS
Subject is stored on successful authentication. If null or empty then
the Subject will not be stored.
-->
<attribute name="SubjectAttributeName">j_subject</attribute>

<attribute name="Config">
  <Server>
    <Service name="JBoss-Tomcat">
      <Engine name="MainEngine" defaultHost="localhost">
        <Logger className="org.jboss.web.tomcat.Log4jLogger"
        verbosityLevel="debug" category="org.jboss.web.localhost.Engine"/>
        <Host name="localhost">

          <!-- Access logger -->
          <Valve className="org.apache.catalina.valves.AccessLogValve"
          prefix="localhost_access" suffix=".log"
          pattern="common" directory="{jboss.server.home.dir}/log"/>

          <!-- This valve clears any caller identity set by the realm
          and provides access to the realm about the existence of an
          authenticated caller to allow a web app to run with a realm
          that support unauthenticated identities. It also establishes
          any run-as principal for the servlet being accessed.
          -->
          <Valve className="org.jboss.web.tomcat.security.SecurityAssociationValve"/>
          <!-- Default context parameters -->
          <DefaultContext cookies="true" crossContext="true" override="true"/>
        </Host>
      </Engine>
    </Service>
  </Server>
</attribute>

```



```
<!-- SSL/TLS Connector configuration -->
<Connector className = "org.apache.coyote.tomcat4.CoyoteConnector"
address="${jboss.bind.address}" port = "8443" scheme = "https" secure
= "true">
  <Factory className =
    "org.apache.coyote.tomcat4.CoyoteServerSocketFactory"
    keystoreFile="${jboss.server.home.dir}/conf/server.keystore"
    keystorePass="tc-ssl"
    protocol = "TLS"/>
</Connector>
</Service>
</Server>
</attribute>
<depends>jboss:service=TransactionManager</depends>
</mbean>

</server>
```

上述介绍的所有方法都可以供开发者使用，本书建议开发者使用 `CoyoteConnector`，因为它是目前较好的连接器。请注意，如果开发者使用第 8 章介绍的 `chap8.keystore` 中的自签名证书测试上述配置，并试图使用 HTTPS 连接访问服务器内容，浏览器会显示警告对话框，即表明浏览器不信任签署了连接到服务器的证书的证书授权中心。比如，当测试第一个配置时，IE 5.5 弹出了安全性警告对话框，如图 9-3 所示。图 9-4 给出了服务器证书详情对话框。这些都和预期的一样，因为自签名证书可以包含任何内容，所以 Web 浏览器遇到类似保护性站点时，应该发出警告。

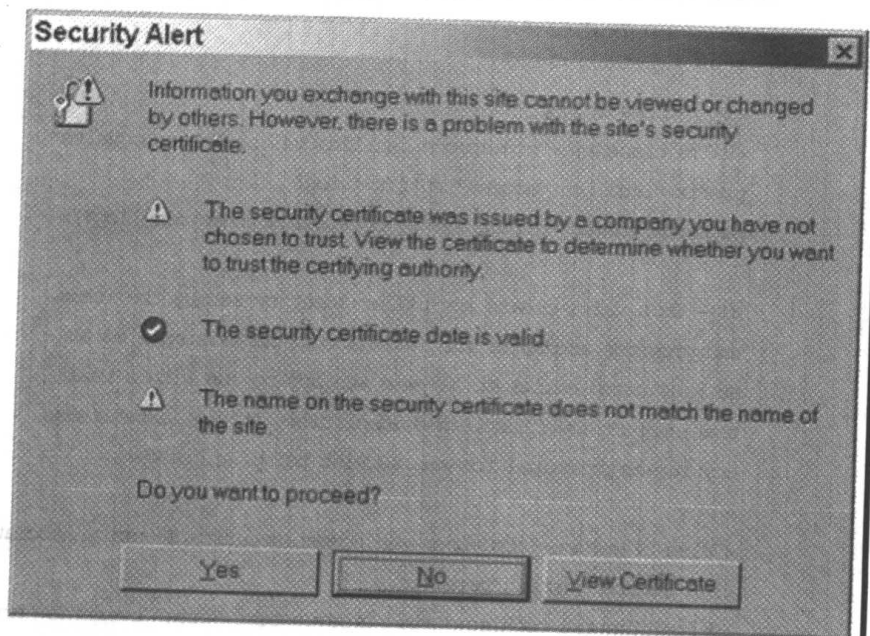


图 9-3 Internet Explorer 5.5 安全性警告对话框

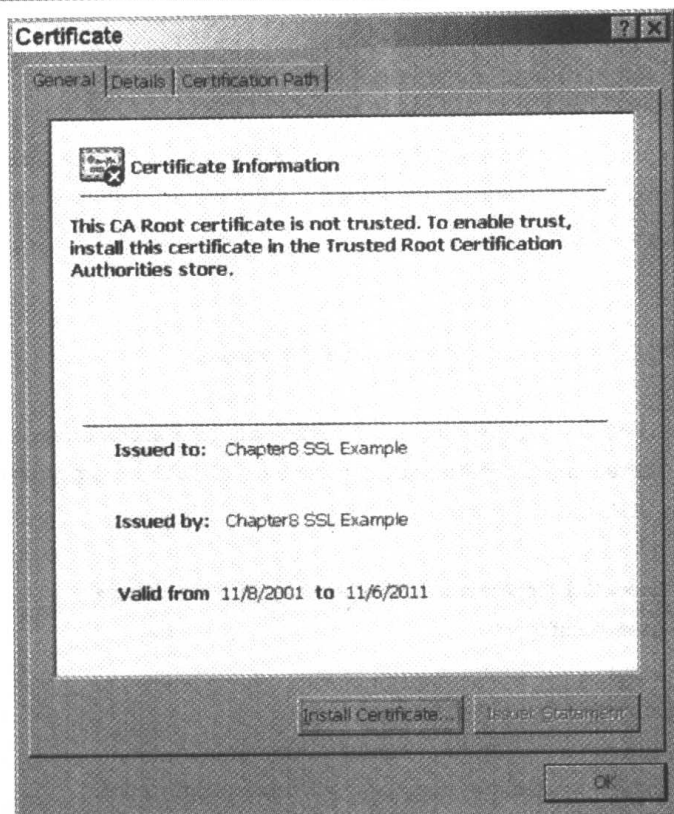


图 9-4 Internet Explorer 5.5 SSL 证书详情对话框

9.2.3 为 JBoss/Tomcat-4.x 绑定配置虚拟主机

自从 JBoss 2.4.5 版发布以来, Servlet 容器层已经对虚拟主机提供了支持。虚拟主机允许根据运行 JBoss 机器的不同 DNS 名分组 Web 应用。比如, 列表 9-8 给出了 jbossweb-tomcat41.sar/META-INF/jboss-service.xml 配置文件实例。该配置文件定义了默认主机“vhost1”和第二个主机“vhost2.mydot.com”。其中, “vhost2.mydot.com”还存在一别名: “www.mydot.com”。

列表 9-8 虚拟主机配置实例

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- An example vhost configuration -->
<server>

    <!-- Get the flag indicating if the normal Java2 parent first class
    loading model should be used over the servlet 2.3 web container first
    model. -->
    <attribute name="Java2ClassLoadingCompliance">true</attribute>
```

```
<attribute name="LenientEjbLink">true</attribute>

<!-- A flag indicating if the JBoss Loader should be used. This loader
uses a unified class loader as the class loader rather than the tomcat
specific class loader.
-->

<attribute name="UseJBossWebLoader">true</attribute>

<!-- The name of the request attribute under with the authenticated JAAS
Subject is stored on successful authentication. If null or empty then
the Subject will not be stored.
-->
<attribute name="SubjectAttributeName">j_subject</attribute>
<attribute name="Config">
  <Server>
    <Service name="JBoss-Tomcat">
      <Engine name="MainEngine" defaultHost="vhost1">
        <Logger className="org.jboss.web.tomcat.Log4jLogger"
        verbosityLevel="debug" category="org.jboss.web.localhost.Engine"/>

        <Host name="vhost1">
          <Alias>vhost1.mydot.com</Alias>

          <Valve className="org.apache.catalina.valves.AccessLogValve"
          prefix="vhost1" suffix=".log"
          pattern="common" directory="{jboss.server.home.dir}/log"/>
          <Valve className="org.jboss.web.tomcat.security.SecurityAssociationValve"/>
          <!-- Default context parameters -->
          <DefaultContext cookies="true" crossContext="true" override="true"/>
        </Host>

        <Host name="vhost2">
          <Alias>vhost2.mydot.com</Alias>
          <Alias>www.mydot.com</Alias>

          <!-- Access logger -->
          <Valve className="org.apache.catalina.valves.AccessLogValve"
          prefix="vhost2" suffix=".log"
          pattern="common" directory="{jboss.server.home.dir}/log"/>
          <Valve className="org.jboss.web.tomcat.security.SecurityAssociationValve"/>
          <!-- Default context parameters -->
          <DefaultContext cookies="true" crossContext="true" override="true"/>
        </Host>
      </Engine>
    </Service>
  </Server>
</attribute>
```



```

<!-- A HTTP/1.1 Connector on port 8080 -->
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
address="${jboss.bind.address}" port="8080" minProcessors="5"
maxProcessors="100" enableLookups="true" acceptCount="10" debug="0"
connectionTimeout="20000" useURIVValidationHack="false"/>
</Service>
</Server>
</attribute>
<depends>jboss:service=TransactionManager</depends>
</mbean>

</server>

```

当部署 WAR 时,默认情况下,该 Web 应用将关联到名字匹配 Engine 元素的 defaultHost 属性值的虚拟主机。为了将 WAR 部署到指定虚拟主机,用户需要使用 jboss-web.xml 描述符和 virtual-host 元素。比如,为了将 WAR 部署到虚拟主机 www.mydot.com 虚拟主机别名上, jboss-web.xml 描述符需要包括如列表 9-9 所示内容,并放置在 WAR WEB-INF 目录中。这也证明除了能够使用 Host 元素的 name 属性值外,也可以使用虚拟主机别名。

列表 9-9 jboss-web.xml 描述符实例 (将 WAR 部署到虚拟主机“www.mydot.com”)

```

<jboss-web>
  <context-root>/</context-root>
  <virtual-host>www.mydot.com</virtual-host>
</jboss-web>

```

9.2.4 使用外部静态内容

开发者可以通过添加外部目录内容到 Tomcat 配置中,从而为 Web 应用使用外部静态文件。比如,为使用“C:/tmp/images”目录中的共享图片,开发者可以编辑 jbossweb-tomcat41.sar/META-INF/jboss-service.xml 描述符,然后添加 Context 入口到 Config 元素中,如列表 9-10 所示。

列表 9-10 包含外部静态内容的配置实例

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- An example tomcat config that includes an external context.

With this you can reference the /images path from any war deployed
to the Host. For example, a test-ex.war index.html page accessible
at http://localhost/test-ex/index.html that includes images from
the external context:

test-ex.war/index.html
<html>

```



```
<body>
  <h1>External Images</h1>
  <ul>
    <li></li>
    <li></li>
    <li></li>
  </ul>
</body>

</html>
-->
<server>

  <mbean code="org.jboss.web.tomcat.tc4.EmbeddedTomcatService"
  name="jboss.web:service=WebServer">
    <attribute name="Java2ClassLoadingCompliance">true</attribute>
    <attribute name="LenientEjbLink">true</attribute>
    <attribute name="UseJBossWebLoader">true</attribute>
    <attribute name="SubjectAttributeName">j_subject</attribute>

  <attribute name="Config">
    <Server>
      <Service name="JBoss-Tomcat">
        <Engine name="MainEngine" defaultHost="localhost">
          <Logger className="org.jboss.web.tomcat.Log4jLogger"
          verbosityLevel="debug" category="org.jboss.web.localhost.Engine"/>
          <Host name="localhost">

            <!-- Access logger -->
            <Valve className="org.apache.catalina.valves.AccessLogValve"
            prefix="localhost_access" suffix=".log"
            pattern="common" directory="{jboss.server.home.dir}/log"/>

            <!-- This valve clears any caller identity set by the realm
            and provides access to the realm about the existence of an
            authenticated caller to allow a web app to run with a realm
            that support unauthenticated identities. It also establishes
            any run-as principal for the servlet being accessed.
            -->
            <Valve className="org.jboss.web.tomcat.security.SecurityAssociationValve"/>
            <!-- Default context parameters -->
            <DefaultContext cookies="true" crossContext="true" override="true">

            <!-- Add a static context /images using directory /tmp/images -->
            <Context path="/images" docBase="/tmp/images" debug="1"
            reloadable="true" crossContext="true">
```

```

        </Context>
    </Host>
</Engine>

<!-- A HTTP/1.1 Connector on port 8080 -->
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
address="{jboss.bind.address}" port="8080" minProcessors="5"
maxProcessors="100" enableLookups="true" acceptCount="10" debug="0"
connectionTimeout="20000" useURValidationHack="false"/>
</Service>
</Server>
</attribute>
<depends>jboss:service=TransactionManager</depends>
</mbean>

</server>

```

请注意 Context 元素的 docBase 属性。该属性通常被平台的 java.io.File 对象看成绝对路径。其含义为，如果在 Win 32 平台上，路径名必须以驱动器盘符开始，比如“C:/tmp/images”。

9.2.5 为 JBoss/Tomcat-4.x 绑定使用 Apache

为将 Apache 作为前端 Web 服务器，并将 Servlet 请求委派给 JBoss/Tomcat 绑定，用户需要在 EmbeddedTomcatService MBean 定义中配置合适的连接器。比如，列表 9-11 给出了配置实例，它用 Apache mod_jk 模块配置使用了 Ajpv13 协议连接器。

列表 9-11 EmbeddedTomcatService MBean 配置实例（使用 Ajpv13 协议连接器集成 Apache）

```

<?xml version="1.0" encoding="UTF-8"?>

<!-- An example AJP configuration -->
<server>

    <mbean code="org.jboss.web.tomcat.tc4.EmbeddedTomcatService"
name="jboss.web:service=WebServer">

        <attribute name="Java2ClassLoadingCompliance">true</attribute>
        <attribute name="LenientEjbLink">true</attribute>
        <attribute name="UseJBossWebLoader">true</attribute>
        <attribute name="SubjectAttributeName">j_subject</attribute>

        <attribute name="Config">
            <Server>
                <Service name="JBoss-Tomcat">
                    <Engine name="MainEngine" defaultHost="localhost">

```

```

<Logger className="org.jboss.web.tomcat.Log4jLogger"
    verbosityLevel="debug" category="org.jboss.web.localhost.Engine"/>
<Host name="localhost">

    <!-- Access logger -->
    <Valve className="org.apache.catalina.valves.AccessLogValve"
        prefix="localhost_access" suffix=".log"
        pattern="common" directory="${jboss.server.home.dir}/log"/>

    <Valve className="org.jboss.web.tomcat.security.SecurityAssociationValve"/>
    <!-- Default context parameters -->
    <DefaultContext cookies="true" crossContext="true" override="true"/>

</Host>
</Engine>

<!-- A AJP 1.3 Connector on port 8009 -->
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
    address="${jboss.bind.address}" port="8009" minProcessors="5"
    maxProcessors="75" enableLookups="true" redirectPort="8443"
    acceptCount="10" debug="0" connectionTimeout="20000"
    useURValidationHack="false"
    protocolHandlerClassName="org.apache.jk.server.JkCoyoteHandler"/>
</Service>
</Server>
</attribute>
<depends>jboss:service=TransactionManager</depends>
</mbean>

</server>

```

其中，将 Tomcat 绑定在 JBoss 中，并不会影响 Apache 的正常配置，即将 Tomcat 绑定在 JBoss 中不会影响 Tomcat 与 Apache 的交互。比如，列表 9-11 给出了 httpd.conf 配置片段，通过它可以测试部署在“/jbosstest”上下文的 WAR：

```

...
LoadModule jk_module libexec/mod_jk.so
AddModule mod_jk.c

<IfModule mod_jk.c>
    JkWorkersFile /tmp/workers.properties
    JkLogFile /tmp/mod_jk.log
    JkLogLevel debug
    JkMount /jbosstest/* ajp13
</IfModule>

```

其他相关 Tomcat 的 Apache 配置同上述模式一样。其中，用户只需要修改 EmbeddedTomcatService MBean 配置中的 Connector 元素定义，便能够满足 Apache 配置要求。

9.2.6 使用群集

自从 JBoss 3.0.1 版发布开始, 嵌入式 Tomcat 服务已支持群集。设置嵌入式 Tomcat 容器的群集功能步骤如下。

步骤

(1) 如果开发者使用了负载均衡器, 则务必设置使用粘性 (sticky) 会话。粘性会话的含义为, 如果用户在节点 A 启动了会话 A, 则只要节点 A 已启动并一直运行, 所有后续请求都必须前向给节点 A。有关配置 Apache Web 服务器粘性会话的更多资料, 请参考 <http://www.ubbeans.com/tomcat/index.html>。

(2) 开发者必须保证目标配置文件集合的 deploy 目录中, 存在 cluster-service.xml 和 jbossha-httpssession.sar 文件, 比如 {JBOSS_HOME}/server/default/deploy 目录。其中, default 配置文件集合中没有包括 cluster-service.xml 文件, 但 JBOSS_DIST/server/all/deploy 目录下存在它。jbossha-httpssession.sar 文件可以通过 JBOSS_DIST/docs/examples/clustering 目录找到。另外, 开发者还需要 jgroups.jar 文件, 通过 JBOSS_DIST/server/all/lib 目录能够找到。

(3) 启动 JBoss, 以验证是否配置好群集功能。首先, 打开 JMX 管理控制台, 即如下 URL: <http://localhost:8080/jmx-console/>。然后, 开发者需要找到如下 MBean, jboss:service=ClusteredHttpSession。其中, “StateString” 内容应该是 “Started”。如果是 “Stopped”, 则请查看服务器的日志文件查明原因。

(4) 启用 Web 应用的群集功能, 即使用 Servlet 2.3 web.xml 描述符声明 distributable 元素。比如:

```
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <distributable/>
  ...
</web-app>
```

(5) 最后, 正常部署 WAR, 则为该 WAR 启用了群集功能。

第 10 章 MBean 服务杂记

本章讨论非常有用的 MBean 服务，其他章节都没有讨论过。因为它们只是实用服务，运行 JBoss 并不需要它们，或者说它们并不适合放入到其他章节中。

10.1 系统属性管理

使用 `org.jboss.varia.property.SystemPropertiesService` MBean 能够管理系统属性。它能够支持设置 JVM 全局属性值，正如 `java.lang.System.setProperty` 方法及“-Dproperty=value”JVM 命令行参数一样。

该 MBean 服务的可配置属性有：

- **Properties:** 使用 `java.util.Properties.load(java.io.InputStream)` 方法格式指定多个属性 “name=value” 对。各个 “name=value” 语句都放置在 Properties 属性元素体的单独行中。
- **URLList:** 用逗号隔开的 URL 字符串列表，MBean 服务将从该列表指定的资源中装入属性文件格式化内容。如果列表的某部分不是 URL，而是相对路径，则 MBean 服务会把它相对于 `<jboss-dist>/server/<conf>` 目录进行解析。比如，“conf/local.properties”，则当运行 default 配置文件集合时，它将作为文件 URL 看待，并指向 `<jboss-dist>/server/default/conf/local.properties`。

列表 10-1 给出了上述两个属性。

列表 10-1 SystemPropertiesService jboss-service.xml 描述符实例

```
<server>
<mbean code="org.jboss.varia.property.SystemPropertiesService"
name="jboss.util:type=Service,name=SystemProperties">

  <!-- Load properties from each of the given comma seperated URLs -->
  <attribute name="URLList">
    http://somehost/some-location.properties,
    ./conf/somelocal.properties
  </attribute>

  <!-- Set propertuies using the properties file style. -->
  <attribute name="Properties">
    property1=This is the value of my property
    property2=This is the value of my other property
  </attribute>

</mbean>
```

</server>

10.2 属性编辑器管理

org.jboss.varia.property.PropertyEditorManagerService MBean 服务能够支持管理 java.bean.PropertyEditor 实例。简单 PropertyEditorManagerService MBean 服务使用 java.bean.PropertyEditorManager 类帮助定义 PropertyEditor。主 (main) jboss-service.xml 文件使用它预装载 (preload) 自定义 JBoss PropertyEditor 实现。某些 JDK 1.3.0 VM 只会从系统类路径中装载 PropertyEditor，因此碰到这种情况就需要使用 PropertyEditorManagerService MBean 服务。

PropertyEditorManagerService MBean 服务支持的属性有：

- **BootstrapEditors**：它是 “property_editor_class=editor_value_type_class” 对列表，定义了 PropertyEditor 到类型的映射。通过 PropertyEditorManager 类的 registerEditor(Class targetType, Class editorClass) 方法能够预装载它。由于该属性的值类型是字符串，因此不需要使用自定义 PropertyEditor 就能从字符串设置它。
- **Editors**：Editors 属性充当了与 BootstrapEditors 同样的功能，但它的类型是 java.util.Properties 类，所以从字符串值设置它需要借助于自定义 PropertyEditor。在能够使用线程上下文类装载器装载自定义 PropertyEditor 的场合，也可以使用该属性，而不是 BootstrapEditors 属性。
- **EditorSearchPath**：该属性允许用户设置 PropertyEditorManager 编辑器包搜索路径。

10.3 服务绑定管理

由于 JBoss 存在很多相互独立的已部署服务，如果在同一台机器上运行多个 JBoss 实例，则编辑配置文件需要浪费大量的时间。JBoss 提供的绑定服务 org.jboss.services.binding.ServiceBindingManager 允许从某中心位置映射服务属性值。在分析了服务描述符文件、并将属性初始值赋给服务后，ServiceConfigurator 会询问 ServiceBindingManager 是否存在需要覆盖的任何内容。实际上，ServicesBindingManager 协调了如下角色之间的关系：ServiceConfigurator、覆盖配置源、服务配置，以及配置委派者。其中，配置委派者知道如何将配置应用于服务。图 10-1 给出了 ServiceBindingManager 中的类关系。

值得开发者关注的一点是，ServiceBindingManager 实现了 JMX MBeanRegistration 接口方法作为它的生命周期通知接口，而不是 JBoss Service 接口。这是必须的，因为 ServiceBindingManager 操作其他服务的属性值。其中，由于属性是在调用任何 JBoss 服务生命周期方法前设置的，因此 ServiceBindingManager 必须在一注册到 MBeanServer 时立即激活。postRegister(Boolean) 回调方法设置了 ServiceBindingManager。

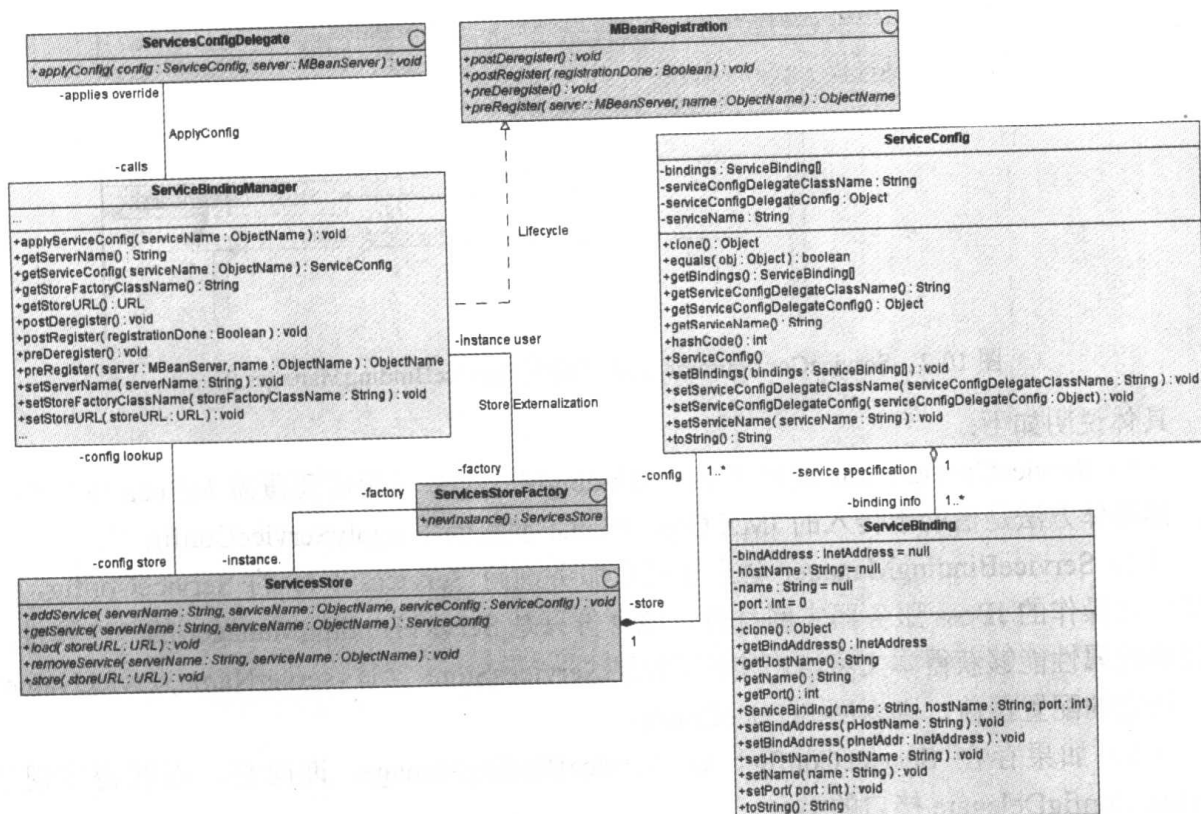


图 10-1 ServiceBindingManager 中 org.jboss.services.binding 包的类图

ServiceBindingManager 通过 ServicesStoreFactory 关联到 ServicesStore。借助于 ServiceBindingManager 的某属性能够设置 ServicesStoreFactory。另外, ServiceBindingManager 的可配置属性集合如下:

- **ServerName:** ServerName 属性指定 ServiceBindingManager 关联的服务器名。ServerName 属性值是逻辑名，ServicesStore 使用它查找 ServiceConfig。
- **StoreFactoryClassName:** StoreFactoryClassName 指定实现 ServicesStoreFactory 接口的类名。开发者可以使用自定义实现，或者使用默认的基于 XML 存储源实现，即 org.jboss.services.binding.XMLServicesStoreFactory。
- **StoreURL:** 配置存储源内容的 URL。ServicesStoreFactory 创建的 ServicesStore 实例中的 load(URL)方法需要使用 StoreURL 属性值。

ServicesStore 仅仅是 ServiceConfig 对象集合，通过 JBoss 实例名和该服务的 JMX ObjectName 能够检索到它。ServiceConfig 是 ServiceBinding 对象集合，并包含 ServicesConfigDelegate。其中，ServicesConfigDelegate 能够将 ServiceBinding 映射到目标 MBean。ServiceConfig 也可能含有 ServicesConfigDelegate 的其他配置信息。ServiceBinding 是 (interface,port) 值对。

当告知 `ServiceBindingManager` 去覆盖服务配置时，具体触发了什么内容呢？图 10-2 给出了答案。

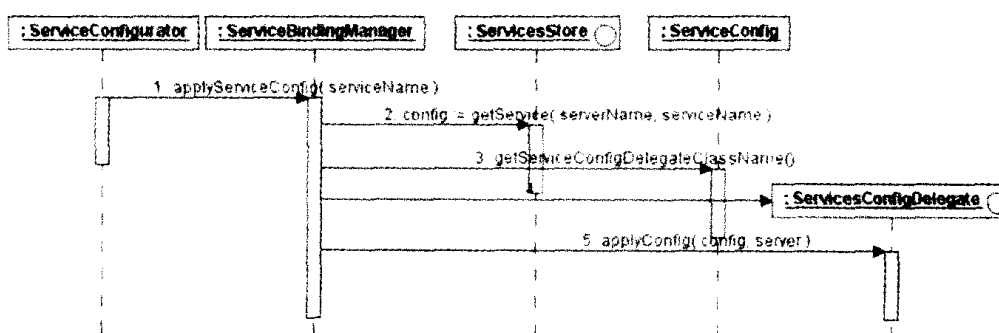


图 10-2 ServiceConfigurator 是如何询问 ServiceBindingManager 的

具体说明如下：

(1) ServiceConfigurator 询问 ServiceBindingManager 是否需要覆盖 MBean 的任何配置。其具体方法是借助于传入的 JMX ObjectName 参数调用 applyServiceConfig 方法。

(2) ServiceBindingManager 替上述指定服务询问 ServicesStore 的 ServiceConfig，从而指定它操作的 JBoss 服务器实例身份。该身份只是 ServiceBindingManager 的一个属性，通过系统属性能够获得本文将给出实例。如果 ServicesStore 含有 <serverName,serviceName> 值对指定的配置覆盖，则返回 ServiceConfig。

(3) 如果存在 ServiceConfig，则 ServiceBindingManager 询问它，以获得实现了 ServicesConfigDelegate 接口的类名。

(4) 使用线程上下文类装载器装载 ServicesConfigDelegate 类，然后创建它。

(5) 请求 ServicesConfigDelegate 实例使用提供的 MBeanServer 应用 Service Config。该委派者将使用 ServiceConfig 信息覆盖指定的服务属性。其中，这些属性是通过调用属性设置器，甚至使用了 MBeanServer 的服务操作指定的。目标服务名存在于 ServiceConfig 中。

这就是 ServiceBindingManager 初步概述。接下来，本文演示如何应用该服务在同一台机器上启动两个使用 default 配置集合服务的 JBoss 实例，从而提供更具体的 ServiceBindingManager 使用演示。

1. 运行两个 JBoss 实例

随 JBoss 发布的服务配置 ServiceBindingManager 和 ServiceStore XML 文件实例能够用于在同一台主机上启动两个 JBoss 实例。至此，本文将浏览启动两个实例的步骤，并查看配置实例。从光盘目录运行如下命令行，能够制作两个服务器配置文件集合 jboss0 和 jboss1：

```

[nr@toki examples]$ ant -Dchap=chap10 -Dex=1 run-example
Buildfile: build.xml
...
run-example1:
    [echo] Preparing jboss0 configuration fileset
    [mkdir] Created dir: /tmp/jboss-3.2.3/server/jboss0
    [copy] Copying 148 files to /tmp/jboss-3.2.3/server/jboss0
    [copy] Copied 2 empty directories to /tmp/jboss-3.2.3/server/jboss0
  
```

```
[copy] Copying 1 file to /tmp/jboss-3.2.3/server/jboss0/conf
[copy] Copying 1 file to /tmp/jboss-3.2.3/server
[echo] Preparing jboss1 configuration fileset
[mkdir] Created dir: /tmp/jboss-3.2.3/server/jboss1
[copy] Copying 148 files to /tmp/jboss-3.2.3/server/jboss1
[copy] Copied 2 empty directories to /tmp/jboss-3.2.3/server/jboss1
```

BUILD SUCCESSFUL

Total time: 8 seconds

上述命令将 server/default 配置文件集合复制为 server/jboss0 和 server/jboss1，并将 ServiceBindingManager 配置生效的 jboss-service.xml 描述符替换了上述两个文件集合中 conf/jboss-service.xml 的任一个。ServiceBindingManager 配置具体如下：

```
<mbean code="org.jboss.services.binding.ServiceBindingManager"
      name="jboss.system:service=ServiceBindingManager">
  <attribute name="ServerName">${jboss.server.name}</attribute>
  <attribute name="StoreURL">${jboss.server.base.dir}/chap10ex1-bindings.xml</attribute>
  <attribute name="StoreFactoryClassName">
    org.jboss.services.binding.XMLServicesStoreFactory
  </attribute>
</mbean>
```

其属性值如下：

- **ServerName:** JBoss 服务器实例的惟一名，用于区分应该将配置覆盖应用于哪个服务器。其中，这里的 {jboss.server.name} 变量引用指配置文件集合目录名。本实例中，它或者为 jboss0，或者为 jboss1。
- **StoreURL:** ServicesStore 配置数据的位置，为 jboss0 和 jboss1 实例定义覆盖内容。{jboss.server.base.dir} 变量引用指向 JBoss server 目录的 URL。本文将 chap10ex1-bindings.xml 安装成实例 1 的配置信息。
- **StoreFactoryClassName:** 默认基于 XML 实现的 ServicesStore。

chap10ex1-bindings.xml 文件含有命名为 jboss0 和 jboss1 的两个服务器配置。其中，jboss0 配置使用默认端口设置，而 jboss1 的各配置端口在 jboss0 基础上分别加 100。该绑定文件复制了 docs/examples/binding-manager/sample-bindings.xml，并将 jboss0 和 jboss1 替换了原有的服务器名。

XMLServicesStoreFactory 类支持图 10-3 给出的 DTD。其中，具体元素如下：

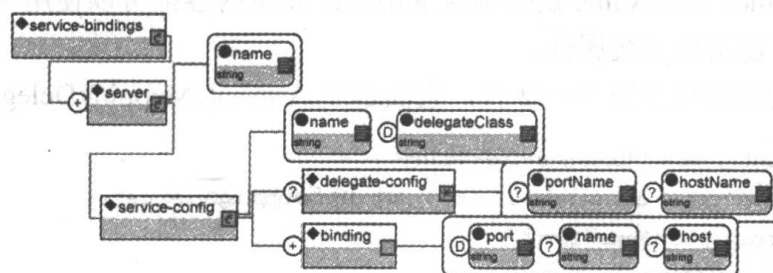


图 10-3 绑定服务 XMLServicesStoreFactory DTD

- **service-bindings**: 配置文件的根元素。它含有一个或多个 **server** 元素。
- **server**: JBoss 服务器实例配置的基位置 (base)。它存在一个 **name** 必须属性, 用于定义 JBoss 实例名。**name** 属性值对应于 **ServiceBindingManager** 的 **ServerName** 属性值。**server** 元素由一个或多个 **service-config** 元素组成。
- **service-config**: 该元素表示用于 MBean 服务的配置覆盖源。它存在一个 **name** 必须属性, 用于表示该配置覆盖源应用的 MBean 服务的 **JMX ObjectName**。它还存在另一个 **delegateClass** 属性, 用于指定 **ServicesConfigDelegate** 实现的类名。其中, 该实现类负责将绑定信息作用于目标 MBean 服务上。**service-config** 元素由一个可选 **delegate-config** 元素和一个或多个 **binding** 元素组成。
- **binding**: 指定端口、地址对。它的可选 **name** 属性能够为服务提供多个绑定。比如, 可以为 Web 容器指定多个虚拟主机。借助于可选 **port** 和 **host** 属性能够分别指定端口和地址。**port** 属性默认值为 0, 即使用匿名端口。**host** 属性默认值为 null, 即使用任何地址。
- **delegate-config**: **delegate-config** 属性指定 **ServicesConfigDelegate** 实现使用的 XML 片段。**hostName** 和 **portName** 属性只是应用于实例中的 **AttributeMappingDelegate**。其中, 将这两个属性放在 **delegate-config** 中的理由在于防止 DTD 编辑器发现它们位于 **AttributeMappingDelegate** 配置中。通常, 这两个属性和 **delegate-config** 的内容都是必须提供的, 但没有较好的方式指定它们, 因为 DTD 中的元素可以存在任意个属性。

目前, JBoss 应用服务器提供的 **ServicesConfigDelegate** 实现有如下两个: **AttributeMappingDelegate** 和 **XSLTConfigDelegate**。其中, **AttributeMappingDelegate** 类实现了 **ServicesConfigDeletate**, 并期望提供如下形式的 **delegate-config** 元素:

```
<delegate-config portName="portAttrName" hostName="hostAttrName">
  <attribute name="someAttrName">someHostPortExpr</attribute>
  ...
</delegate-config>
```

其中, **portAttrName** 是 MBean 服务的属性名, 用于指定绑定端口值; **hostAttrName** 也是 MBean 服务的属性名, 用于指定绑定的主机名。如果没有指定 **portName**, 则没有绑定端口。同理, 如果没有指定 **hostName**, 则没有绑定主机。可选 **attribute** 元素指定由主机和 (或) 端口设置决定的 MBean 属性名。只要在属性内容中引用了 **\${host}** 和 **\${port}**, **AttributeMappingDelegate** 会分别使用绑定的主机和端口替代它们。自从 JBoss 3.2.2RC4 发布开始, **portName**、**hostName** 属性值及 **attribute** 元素内容都可以使用 JBoss 服务描述符支持的 **\${x}** 语法, 以引用系统属性。

比如, 从实例列表摘出第 7~12 行, 它演示了 **AttributeMappingDelegate** 的用法:

```
<service-config name="jboss:service=Naming"
  delegateClass="org.jboss.services.binding.AttributeMappingDelegate">
  <delegate-config portName="Port"/>
  <binding port="1099" />
```

```
</service-config>
```

可以看出, “jboss:service=Naming” MBean 服务的 Port 属性值为 1099。jboss1 服务器配置对应的 Port 属性值为 1199, 它将覆盖 1099。

另外, XSLTConfigDelegate 类实现了 ServicesConfigDelegate, 它期望提供如下形式的 delegate-config 元素:

```
<delegate-config>
  <xslt-config configName="ConfigurationElement">
    <![CDATA[Any XSL document contents...]]>
  </xslt-config>
  <xslt-param name="param-name">param-value</xslt-param>
  ...
</delegate-config>
```

xslt-config 子元素内容指定 XSL 脚本片段, 供该 MBean 服务的 configName 属性使用。而且, 该属性必须是 org.w3c.dom.Element 类型。可选 xslt-param 元素为脚本中使用的参数指定 XSL 脚本参数值。默认情况下, 存在两个参数, 即 host 和 port, 这两个参数值对应于配置中绑定的主机和端口。

XSLTConfigDelegate 能够转换那些使用嵌入 XML 片段配置端口和 (或) 接口的服务。如下内容给出了 jboss1 服务器中的配置信息, 它将 Tomcat Servlet 容器监听端口设置为 8180, 将 AJP 监听端口设置为 8109。

```
<!-- ***** jbossweb-tomcat41.sar ***** -->

<service-config name="jboss.web:service=WebServer"
  delegateClass="org.jboss.services.binding.XSLTConfigDelegate"
>
  <delegate-config>
    <xslt-config configName="Config">
      <![CDATA[
        <xsl:stylesheet
          xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version='1.0'>

          <xsl:output method="xml" />
          <xsl:param name="port"/>

          <xsl:variable name="portAJP" select="$port - 71"/>

          <xsl:template match="/">
            <xsl:apply-templates/>
          </xsl:template>

          <xsl:template match = "Connector">
            <Connector>
              <xsl:for-each select="@*">
```



```

<xsl:choose>
  <xsl:when test="(name() = 'port' and . = '8080')">
    <xsl:attribute name="port"><xsl:value-of select="$port" /></
    xsl:attribute>
  </xsl:when>
  <xsl:when test="(name() = 'port' and . = '8009')">
    <xsl:attribute name="port"><xsl:value-of select="$portAJP" /></
    xsl:attribute>
  </xsl:when>
  <xsl:otherwise>
    <xsl:attribute name="{name()}"><xsl:value-of select="." /></
    xsl:attribute>
  </xsl:otherwise>
</xsl:choose>
</xsl:for-each>
</Connector>
</xsl:template>

<xsl:template match="*|@">
  <xsl:copy>
    <xsl:apply-templates select="@*|node()" />
  </xsl:copy>
</xsl:template>
</xsl:stylesheet>
]]>

</xslt-config>
</delegate-config>
<binding port="8180" />
</service-config>

```

接下来, 本文准备测试配置实例, 即使用先前运行第 10 章中实例 1 创建的 jboss0 和 jboss1 配置文件集合启动这两个 JBoss 实例。开发者通过查看控制台, 能够看到被覆盖的服务端口号。比如, 启动 jboss1 服务器时, 控制台日志如下:

```

2004-05-06 13:39:27,346 INFO [org.jboss.naming.NamingService] Started jnpPort=1299, rmiPort= 1298,
backlog=50, bindAddress=/0.0.0.0, Client SocketFactory=null, Server
SocketFactory=org.jboss.net.sockets. DefaultSocketFactory@ad093076
2004-05-06 13:39:27,445 INFO [org.jboss.naming.NamingService] Listening on port 1299
2004-05-06 13:39:36,152 DEBUG [org.apache.coyote.http11.Http11Protocol] Attribute port: 8280
2004-05-06 13:39:36,208 INFO [org.apache.coyote.http11.Http11Protocol] Initializing Coyote HTTP/1.1
on port 8280
2004-05-06 13:39:36,422 INFO [org.apache.coyote.http11.Http11Protocol] Starting Coyote HTTP/ 1.1 on
port 8280
2004-05-06 13:39:36,645 DEBUG [org.apache.jk.server.JkCoyoteHandler] setProperty port 8209
2004-05-06 13:39:36,900 DEBUG [org.apache.jk.server.JkMain] Substituting port channel Socket.port
8209

```

```
2004-05-06 13:39:36,978 DEBUG [org.apache.jk.server.JkMain] Processing channelSocket::channel Socket
port
2004-05-06 13:39:36,978 DEBUG [org.apache.jk.server.JkMain] Setting port on channelSocket
org.apache.jk.common.ChannelSocket@1ee2de
2004-05-06 13:39:36,978 DEBUG [org.apache.jk.server.JkMain] setProperty org.apache.jk.common.
ChannelSocket@1ee2de port=8209
```

10.4 定时任务

Java 借助于 `java.util.Timer` 和 `java.util.TimerTask` 实用类提供了简单定时器功能。JMX 本身也借助于 `javax.management.timer.TimerMBean` 代理服务，实现了定时 JMX 通知机制，即基于重复时间，间隔处理通知。

JBoss 服务器实现了 JMX 定时器服务的两个新版本，即 `org.jboss.varia.scheduler.Scheduler` 和 `org.jboss.varia.scheduler.ScheduleManager` MBean。这两个 MBean 服务依赖于 JMX 定时器服务，实现了基本的定时功能。它们扩展了 JMX 定时器服务的默认行为，具体如下。

`org.jboss.varia.scheduler.Scheduler`

`Scheduler` 同 `TimerMBean` 的不同之处在于，`Scheduler` 直接调用用户定义类实例的回调方法，或者用户指定 MBean 的操作。

- **InitialStartDate:** 安排的初始调用日期。取值如下：

- NOW: 当前时间 + 1 秒。
- 数字: 即自从 1970 年 1 月 1 日以来的毫秒数。
- Date 的字符串: 表示 `SimpleDateFormat` 将使用默认格式模式“M/d/yy h:mm a”分析它。如果该日期为过去的某个时间，`Scheduler` 将根据初始重复次数和调用周期计算出下次调用时间。其含义为，当重启 MBean（重启 JBoss，等等）时，将会在下次定时时间启动。如果没有启动日期，则 `Scheduler` 永远都不会启动。

比如，如果定时时间安排在每天中午，并且重启了 JBoss 服务器，则下个中午到来时才会启动相应的操作或服务（同理，如果是中午前就会启动，或者如果是中午后，则第二天中午才会启动）。

- **InitialRepetitions:** `Scheduler` 调用目标回调对象的次数。如果为 -1，则除非服务器停止，否则一直重复调用回调对象。
- **StartAtStartup:** 标志位，即当 `Scheduler` 接收到 `startService` 生命周期通知时是否启动。如果为 `true`，则接收到通知立即启动 `Scheduler`。如果为 `false`，则需要显式地调用 `Scheduler` 的 `startSchedule` 操作。
- **SchedulePeriod:** 定时调用周期（单位：毫秒）。`SchedulePeriod` 值必须大于 0。
- **SchedulableClass:** `org.jboss.varia.scheduler.Schedulable` 接口实现的全限定类名，供 `Scheduler` 使用。其中，`Schedulable` 实现的构建器需要使用 `SchedulableArguments` 和 `SchedulableArgumentTypes` 属性值。

- **SchedulableArguments**: 为 **Schedulable** 实现类的构建器提供使用 “,” 隔开的参数列表。其支持的参数有: 原始 (primitive) 数据类型、String 及具有仅能接受单个 String 参数的构建器的类。
- **SchedulableArgumentTypes**: 为 **Schedulable** 实现类的构建器提供使用 “,” 隔开的参数类型列表, 供反射寻找正确的构建器使用。其支持的参数有: 原 (primitive) 数据类型、String 及具有仅能接受单个 String 参数的构建器的类。
- **SchedulableMBean**: 指定被调用的定时 MBean 的全限定 JMX ObjectName 名。如果该 MBean 不可用, 则 Scheduler 不会调用它, 但是还是会消耗剩余重复。当指定 **SchedulableMBean** 时, 同时必须指定 **SchedulableMBeanMethod**。
- **SchedulableMBeanMethod**: 指定定时 MBean 中被调用的操作名。这些方法名可以选择附加上 “(”、逗号隔开的参数关键字及 “)”。支持的参数关键字有:
 - **NOTIFICATION**: 定时器通知实例 (javax.management.Notification) 会替换它。
 - **DATE**: 通知调用日期 (java.util.Date) 会替换它。
 - **REPETITIONS**: 剩余重复次数 (long) 会替换它。
 - **SCHEDULER_NAME**: Scheduler 的 JMX ObjectName 会替换它。
 - 任意全限定类名: Scheduler 会将其设置为 null。

具体 Scheduler 实例只支持单个定时实例。如果需要配置多个定时事件, 则需要使用多个 Scheduler 实例, 并且这些 Scheduler 实例的 JMX ObjectName 都不同。列表 10-2 给出了配置实例, 即 Scheduler 调用了 **Schedulable** 实现及调用 MBean 的配置信息。

列表 10-2 Scheduler jboss-service.xml 描述符实例

```
<server>

  <mbean code="org.jboss.varia.scheduler.Scheduler"
    name="jboss.docs.chap10:service=Scheduler">
    <attribute name="StartAtStartup">true</attribute>
    <attribute name="SchedulableClass">org.jboss.chap10.ex2.ExSchedulable
    </attribute>
    <attribute name="SchedulableArguments">TheName,123456789</attribute>
    <attribute name="SchedulableArgumentTypes">java.lang.String,long
    </attribute>

    <attribute name="InitialStartDate">NOW</attribute>
    <attribute name="SchedulePeriod">60000</attribute>
    <attribute name="InitialRepetitions">-1</attribute>
  </mbean>
</server>
```

列表 10-3 给出了 org.jboss.chap10.ex2.ExSchedulable 实例类的源代码。

列表 10-3 ExSchedulable 类代码

```
package org.jboss.chap10.ex2;

import java.util.Date;
import org.jboss.varia.scheduler.Schedulable;

import org.apache.log4j.Logger;

/** A simple Schedulable example.
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.1 $
 */

public class ExSchedulable implements Schedulable
{
    private static final Logger log = Logger.getLogger(ExSchedulable.class);

    private String name;
    private long value;

    public ExSchedulable(String name, long value)
    {
        this.name = name;
        this.value = value;
        log.info("ctor, name: " + name + ", value: " + value);
    }

    public void perform(Date now, long remainingRepetitions)
    {
        log.info("perform, now: " + now +
            ", remainingRepetitions: " + remainingRepetitions +
            ", name: " + name + ", value: " + value);
    }
}
```

运行如下命名行，部署上述定时器 SAR：

```
[nr@toki examples]$ ant -Dchap=chap10 -Dex=2 run-example
...
run-example2:
[copy] Copying 1 file to /tmp/jboss-3.2.3/server/default/deploy
```

服务器控制台显示了下列信息，这里只是给出前两次定时器调用（调用周期：60 秒）：

```
19:05:01,760 INFO [MainDeployer] Starting deployment of package: file:/private/tmp/jboss-3.2.3/
server/default/deploy/chap10-ex2.sar
19:05:02,047 INFO [ExSchedulable] ctor, name: TheName, value: 123456789
```



```
19:05:02,071 INFO [Scheduler] Started jboss.docs.chap10:service=Scheduler
19:05:02,167 INFO [MainDeployer] Deployed package: file:/private/tmp/jboss-3.2.3/server/default/
deploy/chap10-ex2.sar
19:05:03,050 INFO [ExSchedulable] perform, now: Tue May 04 19:05:03 CDT 2004, remainingRepetitions:
-1, name: TheName, value: 123456789
19:06:03,049 INFO [ExSchedulable] perform, now: Tue May 04 19:06:03 CDT 2004, remainingRepetitions:
-1, name: TheName, value: 123456789
19:07:03,050 INFO [ExSchedulable] perform, now: Tue May 04 19:07:03 CDT 2004, remainingRepetitions:
-1, name: TheName, value: 123456789
```

10.5 JBoss 日志功能框架

JBoss 3.2 的日志功能框架允许使用任何特定框架实现。JBoss 本身使用 `org.jboss.logging.Logger` 作为工厂和日志 (logging) 接口。它同 Log4j `org.apache.log4j.Logger` 一样, 但添加了 trace 日志级别优先权。JBoss 3.2 版本中, `Logger` 类委派给 `LoggerPlugin` 实例, 而不是 JBoss 3.0 中使用的 `org.apache.log4j.Logger`。图 10-4 给出了 `Logger` 和 `LoggerPlugin` 类图。

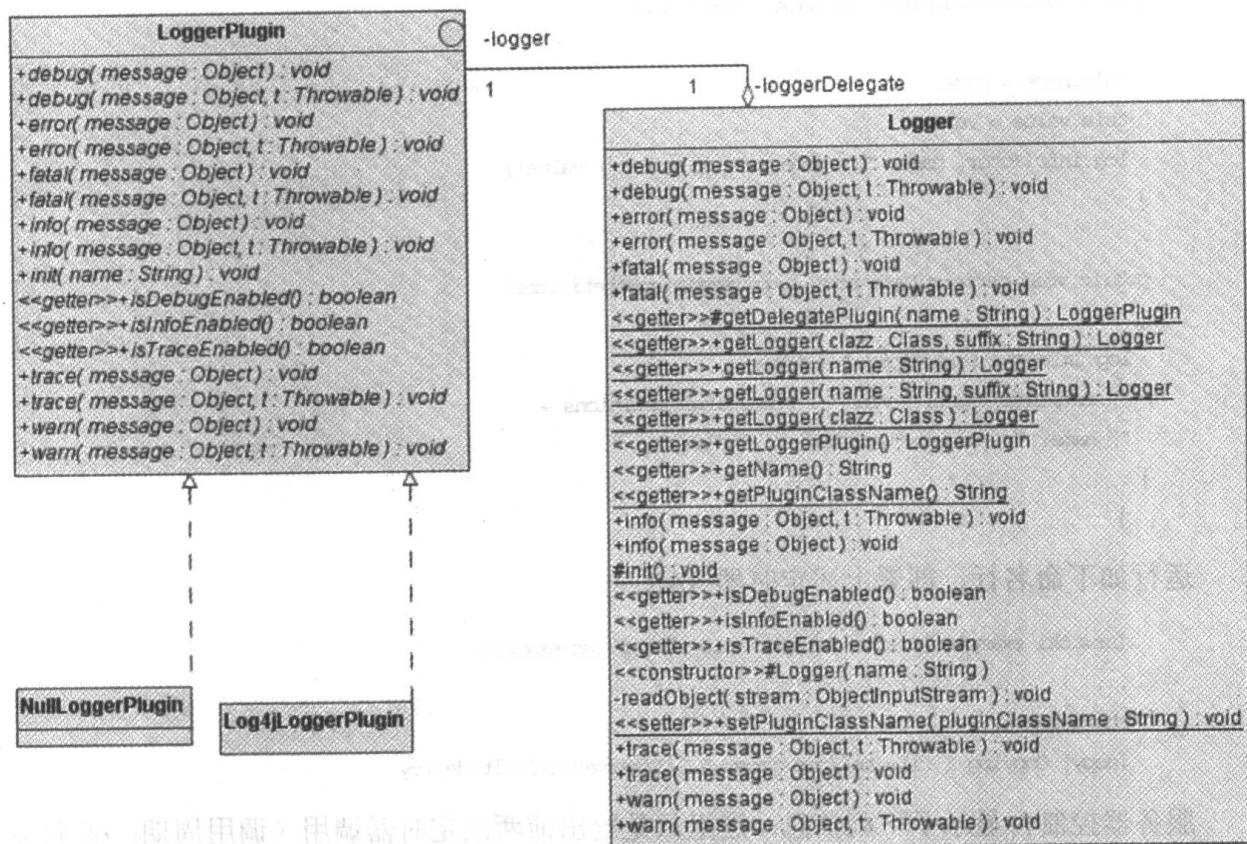


图 10-4 JBoss 日志功能框架类

在默认情况下, JBoss 会继续将 Log4j 框架作为其底层日志功能实现, 即由 `org.jboss`.

logging.Log4jLoggerPlugin 提供的具体实现。为集成其他日志功能实现，开发者需要实现 LoggerPugin 接口，并使用 org.jboss.logging.Logger.pluginClass 系统属性指定实现类名。如果需要失效所有的日志功能，则开发者需要使用 org.jboss.logging.NullLoggerPlugin。NullLoggerPlugin 实现只是简单地提供了 LoggerPlugin 的空方法实现版本。

org.jboss.logging.Log4jService

Log4jService MBean 服务配置 Apache Log4j 系统。JBoss 使用 Log4j 框架作为其内部日志功能 API。

- **ConfigurationURL**: Log4j 配置文件的 URL。用户或者提供 XML 文档，供 org.apache.log4j.xml.DOMConfigurator 使用；或提供 Java 属性文件，供 org.apache.log4j.PropertyConfigurator 使用。URL 内容类型决定了文件类型，如果它为空，则由文件扩展名决定。默认值为“resource:log4j.xml”，即引用活动(active)服务器配置文件集合中的 conf/log4j.xml 文件。
- **RefreshPeriod**: 检查 ConfigurationURL 属性指定的 Log4j 配置是否有变化的周期（单位：秒）。默认值为 60 秒。
- **CatchSystemErr**: 标志位，即如果为 true，则表明 System.err 流应该重定向到 Log4j “STDERR”分类（category）中。默认值为 true。
- **CatchSystemOut**: 标志位，即如果为 true，则表明 System.out 流应该重定向到 Log4j “STDOUT”分类（category）中。默认值为 true。
- **Log4jQuietMode**: Log4jQuietMode 属性为标志位，即如果为 true，则设置 org.apache.log4j.helpers.LogLog.setQuiteMode。由于 Log4j 1.2.8 在 appender 层存在触发死锁的可能性，因此需要设置该属性。更多信息，请参考 bug#696819。

10.6 RMI 动态类装载

org.jboss.web.WebService

WebService MBean 为 RMI 访问服务器 EJB 提供了动态类装载功能。WebService MBean 的可配置属性如下：

- **Port**: WebService 监听端口。如果为 0，则使用任何可用端口。
- **Host**: RMI codebase URL 的 host 部分使用的 public 接口名。
- **BindAddress**: 指定 WebService 监听地址。它能够用于存在多个主机地址的机器上，即为 java.net.ServerSocket 提供监听地址。但只有其中一个地址接受客户请求。
- **Backlog**: 允许连接请求的最大队列长度。如果队列已满且有连接请求时，该请求将被拒绝。
- **DownloadServerClasses**: 标志位，表明当到达的请求没有类装载器键前缀（key prefix）时，服务器是否应该试图从线程上下文类装载器中下载类。

第 11 章 CMP 引擎

本章深入研究 JBoss CMP2 引擎操作，但并不介绍 EJB 2.0 容器管理持久化（Container Managed Persistence, CMP）模型。有关 CMP 2.0 的详细介绍，请开发者参考 J2EE Tutorial¹（具体 URL 为 <http://java.sun.com/j2ee/tutorial/index.html>），或者请开发者参考《Enterprise JavaBeans, 3rd Edition》一书。其中，<http://www.oreilly.com/catalog/entjbeans3/workbooks/index.html> 提供了《Enterprise JavaBeans, 3rd Edition》一书的 JBoss 配套工作手册。

11.1 启 程

JBossCMP 是 EJB 2.0 应用的默认持久化管理器。由于 JBossCMP 是 JBoss 3.x 的核心内容，因此开发者只需要简单地安装 JBoss（请参考 JBoss 3.2 Quick Start Guide）后就能够使用 CMP 2.0。但如果开发者需要创建新的 EJB 2.0 应用，或从 EJB 1.1 应用升级到 EJB 2.0，则需要注意一些技术细节。

当 JBoss 部署 EJB jar 文件时，它会使用 ejb-jar.xml 部署描述符的 DOCTYPE 判断 EJB jar 的版本。列表 11-1 给出了用于 EJB 2.0 的正确 DOCTYPE 表示。

列表 11-1 EJB 2.0 DOCTYPE 声明

```
<!DOCTYPE ejb-jar PUBLIC
"-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">
```

如果 DOCTYPE 的 PUBLIC 标识符是“-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0”，则 JBossCMP 引擎将使用 standardjboss.xml 描述符文件中的“Standard CMP 2.x EntityBean”配置。如果开发者应用中使用了自定义实体 Bean 配置并升级到 EJB 2.0，则必须更换 persistence-manager 元素内容和添加新的拦截器（详情请开发者参考 standardjboss.xml 文件中的“Standard CMP 2.x EntityBean”配置）。除此之外，成功部署和运行 EJB 2.0 应用不再需要开发者完成其他任何配置。

1. 代码实例

本章所有实例的完整源代码位于 examples/src/main/org/jboss/cmp2 目录中。本章待研究对象是犯罪 Portal，下面给出犯罪组织的模型，即图 11-1 给出了犯罪 Portal 的部分数据模型。

¹ 《J2EE Tutorial》第一版的中文版已经由中国铁道出版社于 2003 年翻译出版。

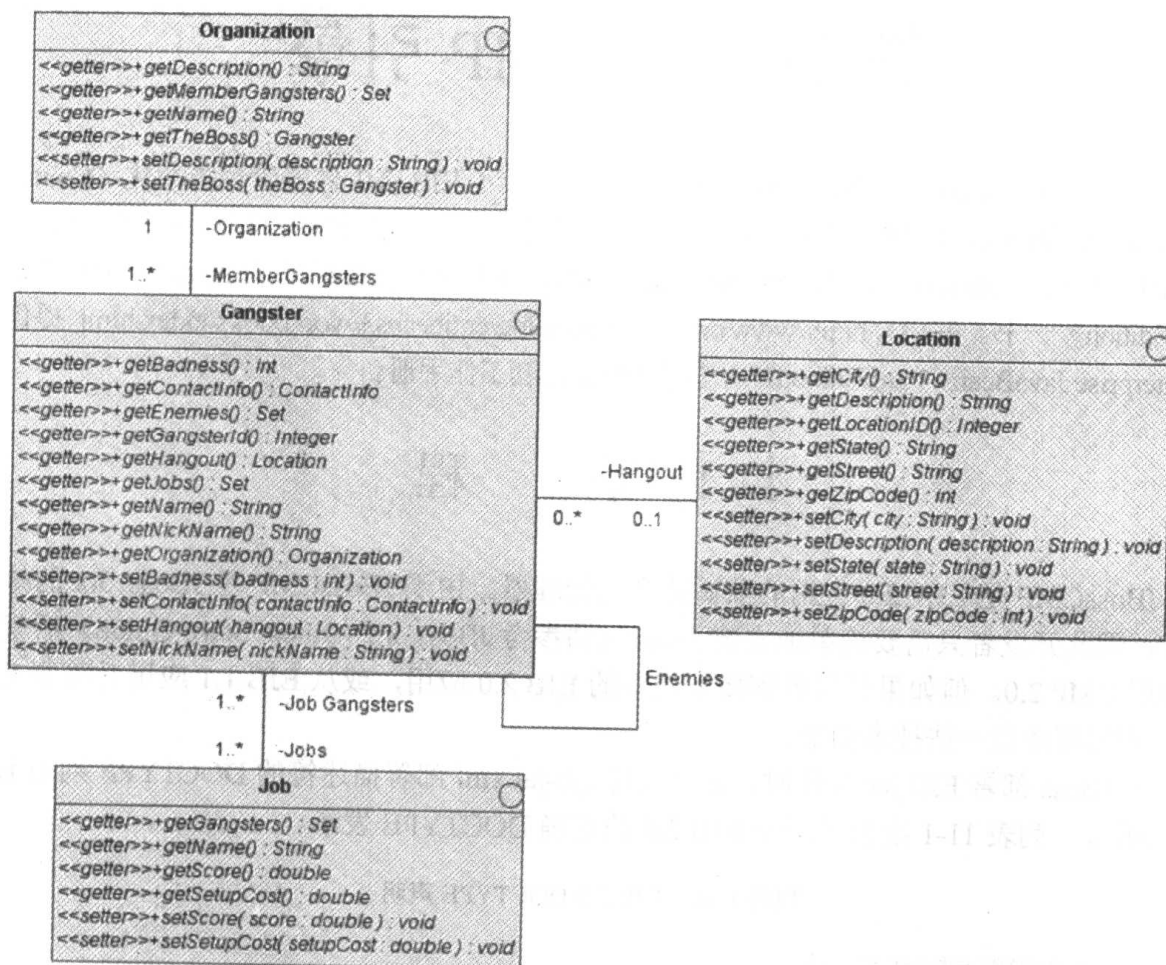


图 11-1 主要的 CMP2 实例类

为完成实例代码的编译工作，请开发者使用如下 Ant 命令：

```

[nr@toki examples]$ ant -Dchap=cmp2 config
Buildfile: build.xml
...
init:
[echo] Using jboss.dist=/tmp/jboss-3.2.3

compile:
[mkdir] Created dir: /Users/orb/Desktop/jboss/docs323/examples/output/classes
[javac] Compiling 123 source files to /Users/orb/Desktop/jboss/docs323/examples/output/classes

config:

prepare:
[mkdir] Created dir: /Users/orb/Desktop/jboss/docs323/examples/output/cmp2
build-ejb.jar:
[jar] Building jar: /Users/orb/Desktop/jboss/docs323/examples/output/cmp2/cmp2-ex1.jar
config:
  
```

```
[copy] Copying 1 file to /tmp/jboss-3.2.3/server/default/deploy
[echo] Waiting for 5 seconds for deploy...
[junit] .
[junit] Time: 4.535
```

```
[junit] OK (1 test)
```

上述命令完成了应用编译和到 JBoss 服务器的部署工作。当开发者启动 JBoss 服务器，或者 JBoss 服务器已经在运行时，则能够浏览到如下部署信息：

```
13:50:18,316 INFO [MainDeployer] Starting deployment of package: file:/private/tmp/jboss-3.2.3/
server/default/deploy/cmp2-ex1.jar
13:50:20,811 INFO [EjbModule] Deploying OrganizationEJB
13:50:20,851 INFO [EjbModule] Deploying GangsterEJB
13:50:20,882 INFO [EjbModule] Deploying JobEJB
13:50:20,918 INFO [EjbModule] Deploying LocationEJB
13:50:20,942 INFO [EjbModule] Deploying EJBTesRunnerEJB
13:50:20,965 INFO [EjbModule] Deploying ReadAheadEJB
13:50:21,590 INFO [EntityInstancePool] Started jboss.j2ee:jndiName=crimeportal/Organization,
plugin=pool,service=EJB
13:50:21,594 INFO [EntityContainer] Started jboss.j2ee:jndiName=crimeportal/Organization,
service=EJB
13:50:21,647 INFO [EntityInstancePool] Started jboss.j2ee:jndiName=crimeportal/Gangster,plugin=
pool,service=EJB
13:50:21,653 INFO [EntityContainer] Started jboss.j2ee:jndiName=crimeportal/Gangster,
service=EJB
13:50:21,721 INFO [EntityInstancePool] Started jboss.j2ee:jndiName=crimeportal/Job,plugin=pool,
service=EJB
13:50:21,725 INFO [EntityContainer] Started jboss.j2ee:jndiName=crimeportal/Job,service=EJB
13:50:21,843 INFO [OrganizationEJB] Created table 'ORGANIZATION' successfully.
13:50:21,972 INFO [GangsterEJB] Created table 'GANGSTER' successfully.
13:50:21,992 INFO [GangsterEJB] Created table 'GANGSTER_ENEMIES' successfully.
13:50:22,169 INFO [JobEJB] Created table 'JOB' successfully.
13:50:22,196 INFO [JobEJB] Created table 'GANGSTER_JOB' successfully.
13:50:22,271 INFO [LocationEJB] Created table 'LOCATION' successfully.
13:50:22,275 INFO [EntityInstancePool] Started jboss.j2ee:jndiName=crimeportal/Location,plugin=
pool,service=EJB
13:50:22,279 INFO [EntityContainer] Started jboss.j2ee:jndiName=crimeportal/Location,
service=EJB
13:50:22,417 INFO [StatelessSessionInstancePool] Started jboss.j2ee:jndiName=ejb/EJBTesRunner,
plugin=pool,service=EJB
13:50:22,431 INFO [StatelessSessionContainer] Started jboss.j2ee:jndiName=ejb/EJBTesRunner,
service=EJB
13:50:22,460 INFO [StatelessSessionInstancePool] Started jboss.j2ee:jndiName=crimeportal/
ReadAhead,plugin=pool,service=EJB
13:50:22,463 INFO [StatelessSessionContainer] Started jboss.j2ee:jndiName=crimeportal/
ReadAhead,service=EJB
```

```
13:50:22,466 INFO [EjbModule] Started jboss.j2ee:module=cmp2-ex1.jar,service=EjbModule
13:50:22,469 INFO [EJBDeployer] Deployed: file:/private/tmp/jboss-3.2.3/server/default/
deploy/cmp2-ex1.jar
13:50:22,591 INFO [MainDeployer] Deployed package: file:/private/tmp/jboss-3.2.3/server/default/
deploy/cmp2-ex1.jar
13:50:38,358 INFO [OrganizationBean$Proxy] Creating organization Yakuza, Japanese Gangsters
13:50:38,506 INFO [OrganizationBean$Proxy] Creating organization Mafia, Italian Bad Guys
13:50:38,514 INFO [OrganizationBean$Proxy] Creating organization Triads, Kung Fu Movie Extras
13:50:38,572 INFO [GangsterBean$Proxy] Creating Gangster 0 'Bodyguard' Yojimbo
13:50:38,651 INFO [GangsterBean$Proxy] Creating Gangster 1 'Master' Takeshi
13:50:38,665 INFO [GangsterBean$Proxy] Creating Gangster 2 'Four finger' Yuriko
13:50:38,697 INFO [GangsterBean$Proxy] Creating Gangster 3 'Killer' Chow
13:50:38,711 INFO [GangsterBean$Proxy] Creating Gangster 4 'Lightning' Shogi
13:50:38,728 INFO [GangsterBean$Proxy] Creating Gangster 5 'Pizza-Face' Valentino
13:50:38,744 INFO [GangsterBean$Proxy] Creating Gangster 6 'Toothless' Toni
13:50:38,785 INFO [GangsterBean$Proxy] Creating Gangster 7 'Godfather' Corleone
13:50:38,809 INFO [JobBean$Proxy] Creating Job 10th Street Jeweler Heist
13:50:38,821 INFO [JobBean$Proxy] Creating Job The Greate Train Robbery
13:50:38,831 INFO [JobBean$Proxy] Creating Job Cheap Liquor Snatch and Grab
```

在运行本章测试程序之前，开发者必须调整 JBossCMP 的日志级别。为生效 CMP 子系统的 DEBUG 级别的日志功能，开发者需要添加如下内容到 log4j.xml 文件中：

```
<category name="org.jboss.ejb.plugins.cmp">
    <priority value="DEBUG"/>
</category>
```

另外，为将 Console Appender 的 Threshold 调整到 DEBUG，使得在控制台能够查看到相关信息，开发者需要对 log4j.xml 文件做如下修改：

```
<!-- ===== -->
<!-- Append messages to the console -->
<!-- ===== -->
<appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">
    <param name="Threshold" value="DEBUG"/>
    <param name="Target" value="System.out"/>
    <layout class="org.apache.log4j.PatternLayout">
        <!-- The default pattern: Date Priority [Category] Message\n -->
        <param name="ConversionPattern" value="%d{ABSOLUTE} %-5p [%c{1}] %m%n"/>
    </layout>
</appender>
```

为了使得开发者能够掌握 JBoss CMP 引擎的完整工作机制，需要将 org.jboss.ejb.plugins.cmp 分类调整到 TRACE 日志级别优先权。开发者需要完成的具体内容如下：

```
<category name="org.jboss.ejb.plugins.cmp">
    <priority value="TRACE" class="org.jboss.logging.XLevel"/>
</category>
```


最后一步，即在具体研究如何运行实例应用之前，开发者还需要注意如下问题。由于在实例配置中将实体 Bean 配置成每次卸载它们时都会删除相应的表，因此每次重启 JBoss 服务器后，开发者都需要再次运行 config 任务，以重新装载实例数据。同时，如果开发者修改了实例，并打算重新部署实例 EJB jar，则开发者也必须使用 config 任务重新装载实例数据。

2. 测试

第一个测试任务是，演示本章即将讨论的各种自定义特性。为使用 Ant 执行相应的任务，请开发者运行如下命令行：

```
[nr@toki examples]$ ant -Dchap=cmp2 -Dex=test run-example
14:03:27,920 DEBUG [OrganizationEJB#findByPrimaryKey] Executing SQL: SELECT name FROM ORGANIZATION
WHERE name=?
14:03:28,011 DEBUG [OrganizationEJB] Executing SQL: SELECT desc, the_boss FROM ORGANIZATION WHERE
(name=?)
14:03:28,020 DEBUG [OrganizationEJB] Executing SQL: SELECT id FROM GANGSTER WHERE (organization=?)
14:03:28,044 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,052 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,070 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,229 DEBUG [GangsterEJB#findBadDudes_ejbql] Executing SQL: SELECT t0_g.id FROM GANGSTER t0_g
WHERE (t0_g.badness > ?)
14:03:28,256 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,264 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,270 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,276 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,281 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,395 DEBUG [GangsterEJB#findBadDudes_jbossql] Executing SQL: SELECT t0_g.id, t0_g.badness
FROM GANGSTER t0_g WHERE (t0_g.badness > ?) ORDER BY t0_g.badness DESC
14:03:28,417 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,423 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,429 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,439 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,446 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,605 DEBUG [GangsterEJB#findBadDudes_declaredsql] Executing SQL: SELECT id FROM GANGSTER WHERE
badness > ? ORDER BY badness DESC
14:03:28,613 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,631 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,641 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,647 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,655 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,783 DEBUG [GangsterEJB#ejbSelectBoss_ejbql] Executing SQL: SELECT DISTINCT
t0_underling_organization_theBos.id FROM GANGSTER t1_underling, ORGANIZATION
t4_underling_organization, GANGSTER t0_underling_organization_theBos WHERE ((t1_underling.name = ?)
OR (t1_underling.nick_name = ?)) AND t1_underling.organization=t4_underling_organization.name AND
t4_underling_organization.the_boss=t0_underling_organization_theBos.id
14:03:28,815 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
```



```
14:03:28,822 DEBUG [GangsterEJB#ejbSelectBoss_ejbql] Executing SQL: SELECT DISTINCT
t0_underling_organization_theBos.id FROM GANGSTER t1_underling, ORGANIZATION
t4_underling_organization, GANGSTER t0_underling_organization_theBos WHERE ((t1_underling.name = ?)
OR (t1_underling.nick_name = ?)) AND t1_underling.organization=t4_underling_organization.name AND
t4_underling_organization.the_boss=t0_underling_organization_theBos.id
14:03:28,829 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:28,835 DEBUG [GangsterEJB#ejbSelectBoss_ejbql] Executing SQL: SELECT DISTINCT
t0_underling_organization_theBos.id FROM GANGSTER t1_underling, ORGANIZATION
t4_underling_organization, GANGSTER t0_underling_organization_theBos WHERE ((t1_underling.name = ?)
OR (t1_underling.nick_name = ?)) AND
t1_underling.organization=t4_underling_organization.name AND t4_underling_organization.the_
boss=t0_underling_organization_theBos.id
14:03:29,011 DEBUG [GangsterEJB#ejbSelectBoss_declaredsql] Executing SQL: SELECT DISTINCT boss.id
FROM GANGSTER boss , gangster g, organization o WHERE (LCASE(g.name) = ? OR LCASE(g.nick_name) = ?)
AND g.organization = o.name AND o.the_boss = boss.id
14:03:29,163 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:29,172 DEBUG [GangsterEJB#ejbSelectBoss_declaredsql] Executing SQL: SELECT DISTINCT boss.id
FROM GANGSTER boss , gangster g, organization o WHERE (LCASE(g.name) = ? OR LCASE(g.nick_name) = ?)
AND g.organization = o.name AND o.the_boss = boss.id
14:03:29,201 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:29,208 DEBUG [GangsterEJB#ejbSelectBoss_declaredsql] Executing SQL: SELECT DISTINCT boss.id
FROM GANGSTER boss , gangster g, organization o WHERE (LCASE(g.name) = ? OR LCASE(g.nick_name) = ?)
AND g.organization = o.name AND o.the_boss = boss.id
14:03:29,378 DEBUG [GangsterEJB#ejbSelectGeneric] DYNAMIC-QL: SELECT OBJECT(g) FROM gangster g WHERE
g.hangout.state IN (?1, ?2, ?3, ?4) ORDER BY g.name
14:03:29,390 DEBUG [GangsterEJB#ejbSelectGeneric] SQL: SELECT DISTINCT t0_g.id, t0_g.name FROM
GANGSTER t0_g, LOCATION t1_g_hangout WHERE (t1_g_hangout.st IN (?, ?, ?, ?) AND
t0_g.hangout=t1_g_hangout.id) ORDER BY t0_g.name ASC
14:03:29,394 DEBUG [GangsterEJB#ejbSelectGeneric] Executing SQL: SELECT DISTINCT t0_g.id, t0_g.name
FROM GANGSTER t0_g, LOCATION t1_g_hangout WHERE (t1_g_hangout.st IN (?, ?, ?, ?) AND
t0_g.hangout=t1_g_hangout.id) ORDER BY t0_g.name ASC
14:03:29,422 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:29,428 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:29,433 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:29,440 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:29,449 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:29,465 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:29,671 DEBUG [OrganizationEJB#ejbSelectOperatingZipCodes_declaredsql] Executing SQL: SELECT
DISTINCT hangout.zip FROM LOCATION hangout , organization o, gangster g WHERE LCASE(o.name) = ? AND
o.name = g.organization AND g.hangout = hangout.id ORDER BY hangout.zip
14:03:29,822 DEBUG [GangsterEJB#ejbSelectGeneric] DYNAMIC-QL: SELECT OBJECT(g) FROM gangster g WHERE
g.hangout.state IN (?1, ?2, ?3, ?4) ORDER BY g.name
14:03:29,828 DEBUG [GangsterEJB#ejbSelectGeneric] SQL: SELECT DISTINCT t0_g.id, t0_g.name FROM
GANGSTER t0_g, LOCATION t1_g_hangout WHERE (t1_g_hangout.st IN (?, ?, ?, ?) AND
t0_g.hangout=t1_g_hangout.id) ORDER BY t0_g.name ASC
14:03:29,832 DEBUG [GangsterEJB#ejbSelectGeneric] Executing SQL: SELECT DISTINCT t0_g.id, t0_g.name
```

```

FROM GANGSTER t0_g, LOCATION t1_g_hangout WHERE (t1_g_hangout.st IN (?, ?, ?, ?) AND
t0_g.hangout=t1_g_hangout.id) ORDER BY t0_g.name ASC
14:03:29,865 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:29,962 DEBUG [GangsterEJB#findByPrimaryKey] Executing SQL: SELECT id FROM GANGSTER WHERE id=?
14:03:29,970 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name, badness, hangout, organization
FROM GANGSTER WHERE (id=?)
14:03:29,980 DEBUG [GangsterEJB] Executing SQL: SELECT cell_area, cell_exch, cell_ext, page_area,
page_exch, page_ext, email FROM GANGSTER WHERE (id=?)
14:03:29,987 DEBUG [GangsterEJB] Executing SQL: UPDATE GANGSTER SET cell_area=?, cell_exch=?,
cell_ext=?, page_area=?, page_exch=?, page_ext=?, email=? WHERE id=?
14:03:29,995 DEBUG [GangsterEJB] Rows affected = 1

```

上述测试内容主要是帮助开发者使用和掌握各种 finder、selector 及 O/R Mapping 问题。本章其余内容会引用这些测试实例进行分析。

3. read-ahead

本章的另外一个主要任务是运行一套测试应用，以证明“11.7 优化装载”节阐述的优化装载配置。至此，本文已正确设置了日志功能，read-ahead 测试将显示查询操作过程中输出的有用信息。请开发者注意，JBoss 服务器能够识别出对 log4j.xml 文件的更改，因此不用重启 JBoss 服务器，但开发者需要等待 1 分钟左右。如下给出 readahead 实际运行的输出信息：

```

[starkam@banshee examples]$ ant -Dchap=cmp2 -Dex=readahead run-example
Buildfile: build.xml
...
run-example:

un-examplereadahead:
[junit] .
[junit] Time: 0.561

[junit] OK (1 test)

```

当执行 readahead 客户应用时，所有执行的 SQL 查询都会显示在 JBoss 服务器控制台中。其中，开发者分析这些输出信息时，要重点关注如下内容：查询次数、选择的列及装载的行数。如下显示出 JBoss 服务器控制台相应的输出结果：

```

#####
### read-ahead none
###
08:31:15,892 DEBUG [findAll_none] Executing SQL: SELECT t0_g.id, t0_g.id FROM GANGSTER t0_g ORDER
BY t0_g.id ASC
08:31:15,902 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name, badness, hangout, organization
FROM GANGSTER WHERE (id=?)
08:31:15,912 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name, badness, hangout, organization
FROM GANGSTER WHERE (id=?)
08:31:15,912 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name, badness, hangout, organization

```

```
FROM GANGSTER WHERE (id=?)
08:31:15,912 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name, badness, hangout, organization
FROM GANGSTER WHERE (id=?)
08:31:15,922 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name, badness, hangout, organization
FROM GANGSTER WHERE (id=?)
08:31:15,922 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name, badness, hangout, organization
FROM GANGSTER WHERE (id=?)
08:31:15,932 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name, badness, hangout,
organization FROM GANGSTER WHERE (id=?)
08:31:15,932 DEBUG [GangsterEJB] Executing SQL: SELECT name, nick_name, badness, hangout, organization
FROM GANGSTER WHERE (id=?)
08:31:15,942 INFO [ReadAheadTest]
###
#####
```

后面讨论优化装载配置时，本文会再次访问该实例，并分析这些输出信息。

11.2 jbosscmp-jdbc 结构

JBossCMP 引擎的行为由 jbosscmp-jdbc.xml 描述符控制。通过如下两种不同方式能够实现其行为的具体控制：第一，使用服务器配置文件集合中 conf/standardjbosscmp-jdbc.xml 描述符实现全局控制；第二，使用各个 EJB jar 部署单元中的 META-INF/jbosscmp-jdbc.xml 描述符实现局部控制。下节阐述 JBossCMP 引擎功能时，会讨论上述描述符中的元素。图 11-2 给出了其顶级（top level）元素，元素具体含义解释如下。

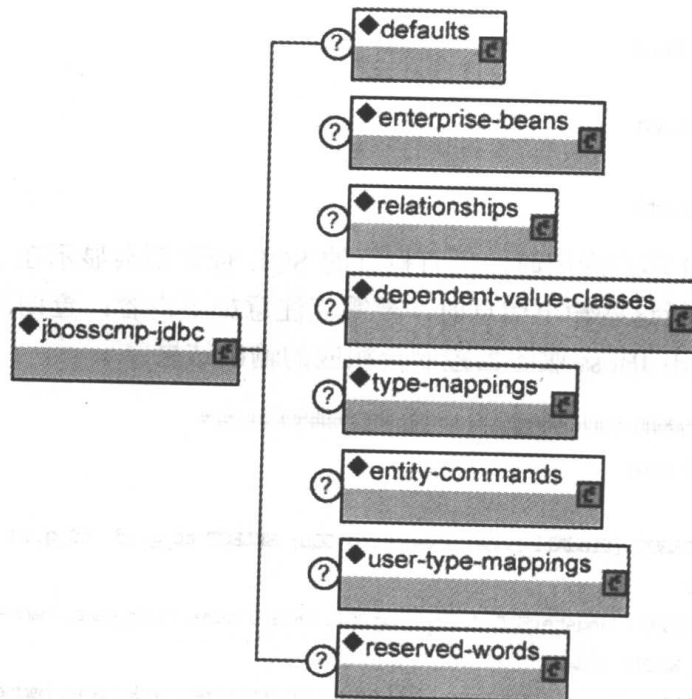


图 11-2 jbosscmp-jdbc 顶级内容模型

- **defaults:** defaults 元素用于制定控制实体 Bean 默认行为的规范。它简化了常见行为的大量设置工作。“11.10 defaults”节内容阐述了 defaults 元素的详细内容。
- **enterprise-beans:** enterprise-beans 元素允许自定义 ejb-jar.xml 描述符中 enterprise-beans 元素定义的实体 Bean。“11.3 实体 Bean”节有其深入分析。
- **relationships:** relationships 元素允许自定义表和实体关系的装载行为。“11.5 容器管理关系”节有其深入分析。
- **dependent-value-classes:** dependent-value-classes 元素允许自定义依赖值类到表的映射。“11.4.6 依赖值类”小节有其深入分析。
- **type-mappings:** type-mappings 元素定义用于具体数据库的 Java 到 SQL 类型映射、SQL 模板及函数映射 (function-mapping)。“11.11 自定义数据源”节有其深入分析。
- **entity-commands:** entity-commands 元素允许定义实体创建命令实例，即如何在持久化存储源中创建实体实例。“11.9 实体命令和主键生成”节有其深入分析。
- **user-type-mappings:** user-type-mappings 元素使用映射器 (mapper) 定义用户类型到列的映射。映射器如同协调者：当需要存储数据时，它拿到用户类型实例，并将实例转换成列值；当需要装载数据时，它拿到列值，并将列值转换为用户类型实例。“11.11.3 用户类型映射”小节内容有其深入分析。
- **reserved-words:** reserved-words 元素定义一个或多个关键字，即创建表时需要转义 (escape) 各个关键字。word 元素内容存储了关键字。

用于 jbossCMP-jdbc.xml 的 DTD

JBOSS_DIST/docs/dtd/jbossCMP-jdbc_3_2.dtd 存放了 jbossCMP-jdbc.xml 描述符使用的 DTD 文件。另外，该 DTD 的 HTML/SVG 版本位于目录：

docs/dtd/html-svg/jbossCMP-jdbc_3_2_dtd/index.html

该 DTD 的 DOCTYPE 声明如下：

```
<!DOCTYPE jbossCMP-jdbc PUBLIC
    "-//JBoss//DTD JBOSSCMP-JDBC 3.2//EN"
    "http://www.jboss.org/j2ee/dtd/jbossCMP-jdbc_3_2.dtd">
```

11.3 实体 Bean

尽管 EJB 2.0 规范为实体 Bean 添加了若干个新特性，并对 cmp-field 和 finder 做了很大的改进，但 CMP 2.0 的基本实体 Bean 结构还是同以前一样。EJB 2.0 引入了新特性，本地接口²。在概念上，这些接口同远程接口和 home 接口（有时候称为，远程 home）相同，但本地接口只具备同一 JVM 内访问能力。因此，本地接口能够使用传址语义，从而不会出现序列化和反序列化方法调用参数³。本地接口不仅仅针对 CMP，这里不给出相关讨论。

² 术语“本地接口”，指单独的 EJBLocalObject，也指 EJBLocalObject/EJBLocalHome 组合。尽管这很容易引起混淆，但这就是当前 EJB 社区的习惯用法。

³ 大部分 J2EE 服务器，包括 JBoss，能够通过使用传址语义优化远程接口中的 JVM 调用。

其中，简化的 Gangster 实体代码如列表 11-2～列表 11-4。

列表 11-2 实例本地 home 接口

```
// Gangster Local Home Interface
public interface GangsterHome extends EJBLocalHome
{
    Gangster create(Integer id, String name, String nickName)
        throws CreateException;
    Gangster findByPrimaryKey(Integer id) throws FinderException;
}
```

列表 11-3 实体本地接口

```
// Gangster Local Interface
public interface Gangster extends EJBLocalObject
{
    Integer getGangsterId();
    String getName();
    String getNickName();
    void setNickName(String nickName);
}
```

列表 11-4 实体实现类

```
// Gangster Implementation Class
public abstract class GangsterBean implements EntityBean
{
    private EntityContext ctx;
    private Category log = Category.getInstance(getClass());

    public Integer ejbCreate(Integer id, String name, String nickName)
        throws CreateException
    {
        log.info("Creating Gangster " + id + " " + nickName + " " + name);
        setGangsterId(id);
        setName(name);
        setNickName(nickName);
        return null;
    }

    public void ejbPostCreate(Integer id, String name,
        String nickName)
    {}

    // CMP field accessors -----
    public abstract Integer getGangsterId();
    public abstract void setGangsterId(Integer gangsterId);

    public abstract String getName();
    public abstract void setName(String name);
}
```

```

public abstract String getNickName();
public abstract void setNickName(String nickName);

public abstract int getBadness();
public abstract void setBadness(int badness);

public abstract ContactInfo getContactInfo();
public abstract void setContactInfo(ContactInfo contactInfo);
...
// EJB callbacks -----
public void setEntityContext(EntityContext context)
{ ctx = context; }
public void unsetEntityContext() { ctx = null; }
public void ejbActivate() { }
public void ejbPassivate() { }
public void ejbRemove() { log.info("Removing " + getName()); }
public void ejbStore() { }
public void ejbLoad() {}
}

```

CMP 2.0 ejb-jar.xml 文件中 entity 的声明方式并没有较大改变。列表 11-5 给出了 GangsterEJB 接口和容器管理持久化域声明。

列表 11-5 ejb-jar.xml entity 声明

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC
    "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
    "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
  <display-name>CMP 2.0 Lab Jar</display-name>

  <enterprise-beans>

    <entity>
      <display-name>Gangster Entity Bean</display-name>
      <ejb-name>GangsterEJB</ejb-name>

      <local-home>org.jboss.cmp2.crimeportal.GangsterHome</local-home>
      <local>org.jboss.cmp2.crimeportal.Gangster</local>
      <ejb-class>org.jboss.cmp2.crimeportal.GangsterBean</ejb-class>

      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
    
```

```

    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>gangster</abstract-schema-name>

    <cmp-field><field-name>gangsterId</field-name></cmp-field>
    <cmp-field><field-name>name</field-name></cmp-field>
    <cmp-field><field-name>nickName</field-name></cmp-field>
    <cmp-field><field-name>badness</field-name></cmp-field>
    <cmp-field><field-name>contactInfo</field-name></cmp-field>

    <primkey-field>gangsterId</primkey-field>
    ...
  </entity>
</enterprise-beans>
</ejb-jar>

```

新的 local-home 和 local 元素等价于 home 和 remote 元素。新 cmp-version 元素值可以为 1.x 或 2.x。添加该元素的目的在于，使得同一 EJB 应用中允许混合 1.x 和 2.x 实体 Bean。新 abstract-schema-name 元素用于标识 EJB-QL 查询中的实体类型。“11.6 查询”节有 abstract-schema-name 元素的深入讨论。

实体映射

JBossCMP 配置实体 Bean 是通过 jbosscomp-jdbc.xml 文件中的 entity 元素完成的。它位于 EJB jar 中的 META-INF 目录里面，该目录还含有用于 JBossCMP 的其他可选配置信息。位于 jbosscomp-jdbc 顶级元素下的 enterprise-beans 元素将 entity 元素组合在其下。列表 11-6 给出了 entity 配置实例。

列表 11-6 jbosscomp-jdbc.xml 实体映射实例

```

<?xml version="1.0" encoding="UTF-8"?>
<DOCTYPE jbosscomp-jdbc PUBLIC
    "-//JBoss//DTD JBOSSCMP-JDBC 3.2//EN"
    "http://www.jboss.org/j2ee/dtd/jbosscomp-jdbc_3_2.dtd">

<jbosscomp-jdbc>
...
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <table-name>gangster</table-name>

      <!-- CMP Fields (请参考“11.4.2 容器管理持久域声明”小节) -->
      <!-- Load Groups (请参考“11.7.2 装载组”小节) -->
      <!-- Queries (请参考“11.6 查询”节) -->
    </entity>
  </enterprise-beans>
</jbosscomp-jdbc>

```

其中, DOCTYPE 声明是可选内容, 如果开发者使用它, 则会减小配置错误发生的几率。另外, 除 `ejb-name` 外, 其他所有元素都是可选的。`ejb-name` 元素用于匹配 `ejb-jar.xml` 文件声明的 `entity` 元素配置。如果没有显示指定, 则所有的默认值都来自于 `jbosscomp-jdbc.xml` 文件的 `defaults` 部分, 或当前服务器配置文件集中的 `conf/standardjbosscomp-jdbc.xml` 文件的 `defaults` 部分。“11.10 defaults”节有 `defaults` 的深入讨论。

图 11-3 为 `entity` 元素内容模型。

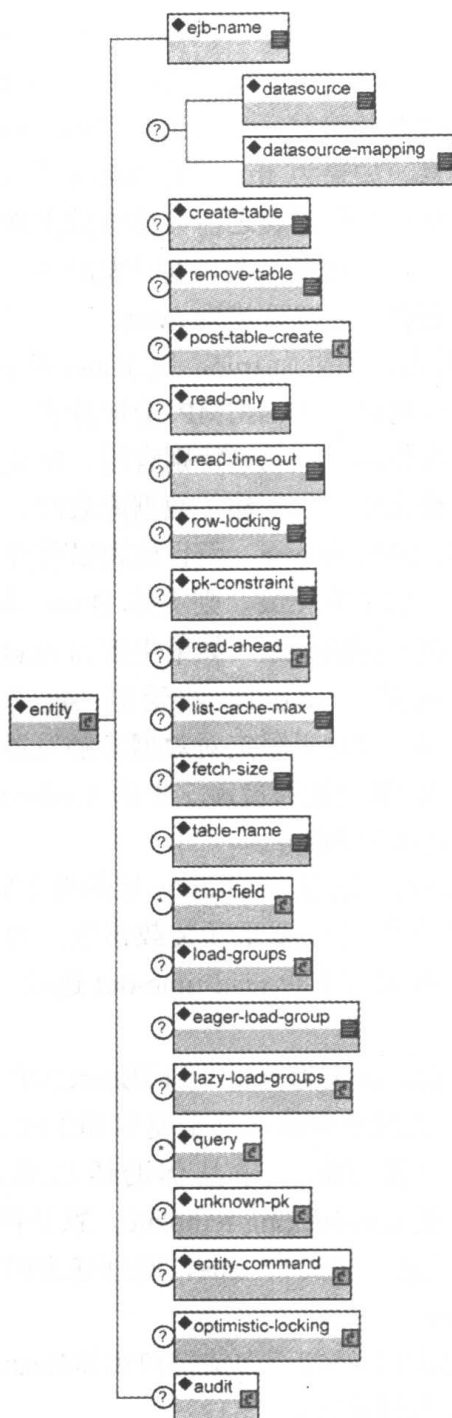


图 11-3 entity 元素内容模型

各个 entity 元素的详细描述如下：

- **ejb-name**: 必需元素，用于指定 EJB 名。该元素必须匹配 ejb-jar.xml 文件中 entity 中的 ejb-name。
- **datasource**: 可选元素，用于指定查找数据源的 jndi-name。实体 Bean 或关系表使用的所有数据库连接都是通过 datasource 元素获得的。本书不推荐实体 Bean 使用不同的数据源，因为这在很大程度上会约束 finder 和 ejbSelect 所能查询的域（domain）范围。默认值为 java:/DefaultDS。
- **datasource-mapping**: 可选元素，用于指定 type-mapping 名。type-mapping 元素用于定义具体数据库的 Java 到 SQL 类型映射、SQL 模板及函数映射。“11.11.2 类型映射”小节会讨论类型映射。默认值为“Hypersonic SQL”。
- **create-table**: 可选元素。如果为 true，则 JBoss 将试图为实体 Bean 创建表。当部署应用时，JBossCMP 在创建表之前会检查数据库表是否存在。如果找到，则记录到日志中，然后不会创建表。在早期开发阶段，由于表结构经常变动，因此在这种场合该元素特别有用。默认值为 false。
- **remove-table**: 可选元素。如果为 true，则 JBoss 将试图删除实体 Bean 的表和相应的关系表。当卸载应用时，JBossCMP 会删除表。在早期开发阶段，由于表结构经常变动，因此在这种场合该元素特别有用。默认值为 false。
- **post-table-create**: 可选元素。当创建了数据库表时，立即执行该元素指定的 SQL 语句。但前提是 create-table 为 true，而且该表以前并不存在。
- **read-only**: 可选元素。如果为 true，则实体 Bean 供应商不能改变任何域的值。只读域不允许存储或插入到数据库。如果主键为 read-only，则 create 方法将抛出 CreateException。如果某 read-only 域受到 set 访问方法的调用，则抛出 EJBException。read-only 域对于限定对数据库触发器填入值，比如最近更新的修改很有用。在 cmp-field 级别能够覆盖这里的 read-only 选项，“11.4.4 read-only 域”小节有介绍。默认值为 false。
- **read-time-out**: 可选元素。读取 read-only 域的有效时间（单位：毫秒）。如果为 0，则表示每次事务开始时总是要重新装载该值。如果为 -1，则该值从不超时。在 cmp-field 级别能够覆盖这里的 read-time-out 选项。如果为 false，则忽略该值。默认值为 -1。
- **row-locking**: 可选元素。如果为 true，则 JBossCMP 将锁定事务装入的所有行。当装入实体 Bean 时，大部分数据库都是使用 SELECT FOR UPDATE 语法实现这里的行锁行为的，但是 JBoss 中具体的语法要取决于实体 Bean 使用的 datasource-mapping 中的 row-locking-template。默认值为 false。
- **pk-constraint**: 可选元素。如果为 true，则创建表时，JBossCMP 会为表添加主键约束。默认值为 true。
- **read-ahead**: 可选元素，用于控制查询结果和实体 Bean 的 cmr-field 的缓存。“11.7.3 read-ahead”小节将深入讨论它。
- **list-cache-max**: 可选元素，用于指定实体 Bean 缓存区中的列表数量。11.7.3 小节中的“on-load”有其讨论。默认值为 1 000。

- **fetch-size**: 可选元素, 指定一次性从底层数据存储源中读入实体 Bean 的数量。默认值为 0。
- **table-name**: 可选元素, 指定用于存储实体 Bean 数据的表名。每个实体 Bean 实例将存储为表的单行数据。默认值为 ejb-name 元素值。
- **cmp-field**: 可选元素, 指定如何将 ejb-jar.xml 中 cmp-field 映射到持久化存储源。“11.4 容器管理持久域”节将有讨论。
- **load-groups**: 可选元素, 指定 cmp-field 的一个或多个分组 (grouping), 以声明域的分组装载。“11.7.2 装载组”小节有其讨论。
- **eager-load-groups**: 可选元素, 将一个或多个分组装载定义为 eager 装载组。11.7.4 小节中的“2. eager 装载过程”有阐述。
- **lazy-load-groups**: 可选元素, 将一个或多个分组装载定义为 lazy 装载组。11.7.4 小节中的“3. lazy 装载过程”有阐述。
- **query**: 可选元素, 定义 find 和 ejbSelect 方法。“11.6 查询”节有阐述。
- **unknown-pk**: 可选元素, 定义如何将未知主键类型 (*java.lang.Object*) 映射到持久化存储源。
- **entity-command**: 可选元素, 定义实体 Bean 生成命令实例。通常, 开发者使用它定义自定义命令实例, 供主键生成使用。“11.9 实体命令和主键生成”节有相关讨论。
- **optimistic-locking**: 可选元素, 定义乐观锁策略。“11.8 乐观锁”节有深入讨论。
- **audit**: 可选元素, 指定待评审的容器管理持久化域。“11.4.5 评审实体 Bean 访问”小节有深入讨论。

11.4 容器管理持久域

11.4.1 容器管理持久域抽象访问方法

尽管 CMP 2.0 在功能上并没有改动容器管理持久域, 但是不再通过企业 Bean 实现类声明域。在 CMP 2.0 中, 容器管理持久域不能够直接访问, 而是将各个容器管理持久域使用抽象访问方法集合声明在实体 Bean 实现类中。抽象访问方法类似于 JavaBean 属性访问方法, 但容器管理持久域的抽象访问方法不提供实现。比如, 列表 11-7 声明了 gangster 实体 Bean 中的 gangsterId、name、nickName 及 badness 容器管理持久域访问方法。

列表 11-7 cmp-field 抽象访问方法声明实例

```
public abstract class GangsterBean implements EntityBean
{
    public abstract Integer getGangsterId();
    public abstract void setGangsterId(Integer gangsterId);
    public abstract String getName();
}
```

```
public abstract void setName(String param);
public abstract String getNickName();
public abstract void setNickName(String param);
public abstract int getBadness();
public abstract void setBadness(int param);
}
```

开发者必须为每个容器管理持久域同时提供 getter 和 setter 方法，每个访问方法必须声明为 public abstract。

11.4.2 容器管理持久域声明

EJB 2.0 ejb-jar.xml 文件的 cmp-field 声明仍然没有改变。比如，为列表 11-7 声明的容器管理持久域而提供的 ejb-jar.xml 文件如列表 11-8 所示。

列表 11-8 ejb-jar.xml cmp-field 声明

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <cmp-field><field-name>gangsterId</field-name></cmp-field>
      <cmp-field><field-name>name</field-name></cmp-field>
      <cmp-field><field-name>nickName</field-name></cmp-field>
      <cmp-field><field-name>badness</field-name></cmp-field>
    </entity>
  </enterprise-beans>
</ejb-jar>
```

11.4.3 容器管理持久域列映射

对于实体 Bean 而言，ejb-jar.xml 中 cmp-field 的映射由 jbosscmp-jdbc.xml 中的 cmp-field 元素声明。jbosscmp-jdbc.xml 的 cmp-field 元素的内容模型如图 11-4 所示。

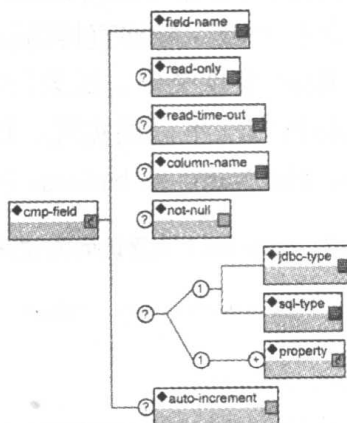


图 11-4 jbosscmp-jdbc.xml cmp-field 元素内容模型

列表 11-9 给出了 cmp-field 映射的使用实例。

列表 11-9 jbosscmp-jdbc.xml cmp-field 映射

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <table-name>gangster</table-name>

      <cmp-field>
        <field-name>gangsterId</field-name>
        <column-name>id</column-name>
      </cmp-field>
      <cmp-field>
        <field-name>name</field-name>
        <column-name>name</column-name>
        <not-null/>
      </cmp-field>
      <cmp-field>
        <field-name>nickName</field-name>
        <column-name>nick_name</column-name>
        <jdbc-type>VARCHAR</jdbc-type>
        <sql-type>VARCHAR(64)</sql-type>
      </cmp-field>
      <cmp-field>
        <field-name>badness</field-name>
        <column-name>badness</column-name>
      </cmp-field>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

在 cmp-field 元素中，用户可以控制列名和数据类型。cmp-field 元素的详细描述如下：

- **field-name:** 必需元素，cmp-field 名。field-name 必须匹配 ejb-jar.xml 文件中实体 Bean 对应 cmp-field 的 field-name 元素。
- **column-name:** 可选元素，cmp-field 映射的列名。默认值为 field-name 属性值。
- **not-null:** 可选元素。当 JBossCMP 自动为实体 Bean 创建表时，它应该添加 not null 到列声明后面。默认情况下，将主键和原始类型设置为 not null。
- **jdbc-type:** 当设置 JDBC PreparedStatement 中的参数或从 JDBC ResultSet 装载数据时，开发者才需要使用 JDBC 类型。java.sql.Types 定义了合法类型。只有在指定了 sql-type 时才需要该元素。默认时，其取值取决于 datasource-mapping。
- **sql-type:** 供创建表语句使用的 SQL 类型。只有数据库厂商才能够决定合法的 SQL 类型。只有在指定了 jdbc-type 时才需要该元素。默认时，其取值取决于 datasource-mapping。

- **property**: 可选元素，定义如何将依赖值类 `cmp-field` 属性映射到持久化存储源。“11.4.6 依赖值类”小节有更进一步的讨论。
- **auto-increment**: 可选元素，表明它在数据库层是自动增加的，供映射域到生成的列和外部操作列使用。
- **dbindex**: 可选元素，表明服务器应该为数据库中对应列创建索引，并且索引名是“<fieldname>_index”。

11.4.4 read-only 域

`cmp-field` 使用抽象访问方法的另一个优势在于，它能够提供只读（read-only）域。对于 1.x CMP 引擎，即 JAWS 而言，只读是通过实体 Bean 的 `read-only` 和 `read-time-out` 支持的。然而，在 CMP 1.x 中实体 Bean 供应商总是能够改变只读实体 Bean 的域值，容器却无能为力。对于 CMP 2.x 而言，容器供应商提供了访问方法的实现，因此当实体 Bean 供应商试图修改只读实体 Bean 的域值时，容器会抛出异常。

JBossCMP 将只读特性扩展到域级别，即能够将 `read-only` 和 `read-time-out` 元素作用于 `cmp-field` 元素。这些元素工作机制同实体 Bean 级别一样。如果某域为只读，则它决不会用在 INSERT 或 UPDATE 语句中。如果某主键域为只读，则调用 `create` 方法容器会抛出 `EJBException`。比如最近更新中，对于由数据库触发器填充的域，`read-only` 特别有用。列表 11-10 给出了 `cmp-field` 中声明 `read-only` 的实例。

列表 11-10 jbosscmp-jdbc.xml `cmp-field read-only` 声明实例

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <cmp-field>
        <field-name>lastUpdated</field-name>
        <read-only>true</read-only>
        <read-time-out>1000</read-time-out>
      </cmp-field>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

11.4.5 评审实体 Bean 访问

`entity` 中的 `audit` 元素允许用户指定如何访问和评审实体 Bean。但前提是开发者必须在安全性域下访问实体 Bean，因此建立了调用者身份。图 11-5 给出了 `audit` 元素的内容模型。列表 11-11 给出了 `audit` 元素声明实例。

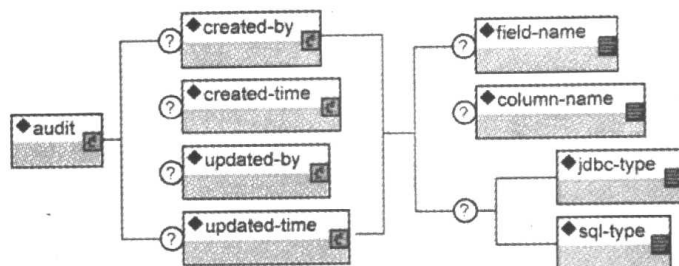


图 11-5 jboss-cmp-jdbc.xml audit 元素的内容模型

- **created-by**: 可选元素, 表明创建实体 Bean 的调用者应该被存入到指定的 column-name 或 cmp field-name 中。
- **created-time**: 可选元素, 表明创建实体 Bean 的时间应该被存入到指定的 column-name 或 cmp field-name 中。
- **updated-by**: 可选元素, 表明最近更新实体 Bean 的调用者应该被存入到指定的 column-name 或 cmp field-name 中。
- **updated-time**: 可选元素, 表明最近修改实体 Bean 的时间应该被存入到指定的 column-name 或 cmp field-name 中。
- ***/field-name**: 该元素表明对应的评审信息应该保存到被访问实体 Bean 指定的 cmp-field 中。请注意, 不一定要求实体 Bean 中存在相匹配的 cmp 域。如果匹配到域名, 则能够在应用中使用 cmp 域的抽象 getter 和 setter 访问方法访问评审信息。否则, 将创建评审域, 并添加到实体 Bean 中。用户能够使用评审域名, 并借助于 EJB-QL 查询访问评审信息, 而不能够直接通过实体 Bean 访问方法。
- ***/column-name**: 该元素表明对应的评审信息应该保存在实体 Bean 表指定的列中。如果 JBossCMP 创建了数据库表, 则可以使用 jdbc-type 和 sql-type 元素定义存储类型。

列表 11-11 audit 元素声明实例

```

<jboss-cmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>AuditChangedNamesEJB</ejb-name>
      <table-name>cmp2_audit_changednames</table-name>
      <audit>
        <created-by>
          <column-name>createdby</column-name>
        </created-by>
        <created-time>
          <column-name>createdtime</column-name>
        </created-time>
        <updated-by>
          <column-name>updatedby</column-name></updated-by>
        <updated-time>
          <column-name>updatedtime</column-name>
        </updated-time>
      </audit>
    </entity>
  </enterprise-beans>
</jboss-cmp-jdbc>

```

```
</updated-time>
</audit>
</entity>
</enterprise-beans>
</jboss-cmp-jdbc>
```

11.4.6 依赖值类

假想术语，即依赖值类（Dependent Value Class, DVC），用于标识 cmp-field 类型的任何 Java 类。请开发者注意，JBossCMP 能够自动识别的类型不属于 DVC 所标识的范围内。更多信息，请开发者参考 Enterprise JavaBeans 规范 Version 2.0 最终发布版。默认情况下，DVC 是序列化的，并且这种序列化形式能够存储在数据库的单列中。长期存储序列化形式的类是经常讨论的主题，本文不给出讨论。JBossCMP 支持将 DVC 内部数据存入到一个或多个列中，这对于支持遗留 JavaBean 或数据库结构非常有用。开发者很难找到提供高度平面化的（flattened）结构（比如，PURCHASE_ORDER 表含有 SHIP_LINE1、SHIP_LINE2、SHIP_CITY 等域，以及适合于送单地址的一套属性）的数据库。其他常见数据库结构包括：含有邮编、传真等单独域的电话号码，含有若干个域的人名。对于 DVC 而言，能够将多列映射到单个逻辑 JavaBean 中。请注意，DVC 和依赖值对象⁴（Dependent Value Object）不相同。

JBossCMP 要求待映射的 DVC 必须遵循 JavaBean 命名规范，并使用简单属性，各个待存储到数据库的属性必须同时提供 setter 和 getter 方法⁵。另外，DVC 必须是可序列化的，并无参数构造器。属性可以是任何简单类型，未映射的 DVC 或已映射的 DVC，但不能是 EJB⁶。在 dependent-value-classes 元素中指定 DVC 映射，图 11-6 显示了 dependent-value-classes 元素的内容模型。

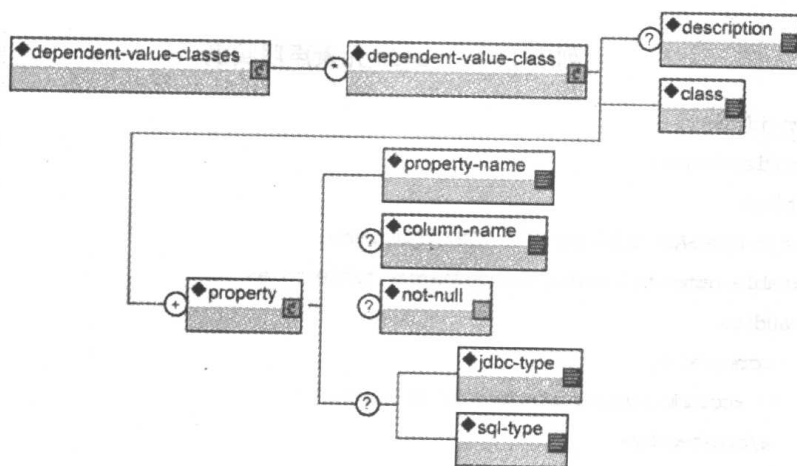


图 11-6 jboss-cmp-jdbc.xml 依赖值类声明

⁴ EJB 2.0 规范建议最终草案 1 新增了依赖值对象，但随后被建议最终草案 2 中的本地接口替代。

⁵ JBossCMP 后续发布版将对如下需求不做要求：DVC 使用 JavaBean 命名规范。

⁶ JBossCMP 后续发布版也将删除该限制。目前的建议是，允许通过无参数方法获得属性值，允许通过单参数方法或构造器设置属性值。

列表 11-12 给出了电话号码 DVC、联系信息 DVC 声明实例，以及对应类代码。

列表 11-12 jbosscomp-jdbc.xml 依赖值类声明

```
<jbosscomp-jdbc>
  <dependent-value-classes>
    <dependent-value-class>
      <description>A phone number</description>
      <class>org.jboss.comp2.crimeportal.PhoneNumber</class>
      <property>
        <property-name>areaCode</property-name>
        <column-name>area_code</column-name>
      </property>
      <property>
        <property-name>exchange</property-name>
        <column-name>exchange</column-name>
      </property>
      <property>
        <property-name>extension</property-name>
        <column-name>extension</column-name>
      </property>
    </dependent-value-class>

    <dependent-value-class>
      <description>General contact info</description>
      <class>org.jboss.comp2.crimeportal.ContactInfo</class>
      <property>
        <property-name>cell</property-name>
        <column-name>cell</column-name>
      </property>
      <property>
        <property-name>pager</property-name>
        <column-name>pager</column-name>
      </property>
      <property>
        <property-name>email</property-name>
        <column-name>email</column-name>
        <jdbc-type>VARCHAR</jdbc-type>
        <sql-type>VARCHAR(128)</sql-type>
      </property>
    </dependent-value-class>
  </dependent-value-classes>
</jbosscomp-jdbc>

public class PhoneNumber implements Serializable {
  /** The first three digits of the phone number. */
  private short areaCode;
```



```
/** The middle three digits of the phone number. */
private short exchange;

/** The last four digits of the phone number. */
private short extension;
...
}

public class ContactInfo implements Serializable {
    /** The cell phone number. */
    private PhoneNumber cell;

    /** The pager number. */
    private PhoneNumber pager;

    /** The email address */
    private String email;
    ...
}
```

每个 DVC 都是通过 `dependent-value-class` 元素声明的。`class` 元素声明的 Java 类型标识了 DVC。每个持久化属性借助于 `property` 元素声明，`property` 元素必须基于 `cmp-field` 元素，因此不需要对 `property` 元素加以说明。JBossCMP 后续发布版将会把上述约束删除掉。目前，本书建议：如果是本地实体 Bean，则存储主键域；如果是远程实体 Bean，则存储实体 Bean Handle。

`dependent-value-classes` 定义内部结构和类的默认映射。当 JBossCMP 遇到未知类型的域，则会搜索注册 DVC 列表。如果找到，则将域存储到列集合中；如果没有找到，则将它以序列化形式存储到单列中。JBossCMP 不支持 DVC 继承，因此上述只搜索域的声明类型。同时，可以使用 DVC 构建其他 DVC，所以当 JBossCMP 运行到这种 DVC 时，它会将 DVC 树结构转换成列集合。如果在启动期间，JBossCMP 发现存在 DVC 回路，则抛出 `EJBException`。`property` 的默认列名是对应 `cmp-field` 的列名。如果 `property` 是 DVC，则重复上述过程。比如，`ContactInfo` 类型的 `cmp-field` 和指定 `info`（请参考列表 11-12），JBossCMP 将它们转换为如列表 11-13 所示。

列表 11-13 为 `ContactInfo` 依赖值类生成的列名

```
info_cell_area_code
info_cell_exchange
info_cell_extension
info_pager_area_code
info_pager_exchange
info_pager_extension
info_email
```

自动生成的列名很长，而且不易于使用。在 `entity` 元素中能够覆盖列的默认映射，见

列表 11-14。

列表 11-14 jbosscmp-jdbc.xml cmp-field 依赖值类 property 覆盖

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <cmp-field>
        <field-name>contactInfo</field-name>
        <property>
          <property-name>cell.areaCode</property-name>
          <column-name>cell_area</column-name>
        </property>
        <property>
          <property-name>cell.exchange</property-name>
          <column-name>cell_exch</column-name>
        </property>
        <property>
          <property-name>cell.extension</property-name>
          <column-name>cell_ext</column-name>
        </property>
        <property>
          <property-name>pager.areaCode</property-name>
          <column-name>page_area</column-name>
        </property>
        <property>
          <property-name>pager.exchange</property-name>
          <column-name>page_exch</column-name>
        </property>
        <property>
          <property-name>pager.extension</property-name>
          <column-name>page_ext</column-name>
        </property>
        <property>
          <property-name>email</property-name>
          <column-name>email</column-name>
          <jdbc-type>VARCHAR</jdbc-type>
          <sql-type>VARCHAR(128)</sql-type>
        </property>
      </cmp-field>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

正如列表 11-14 显示的一样，当覆盖 entity 元素的 info 属性时，需要从平面角度引用属性，比如 cell.areaCode。

11.5 容器管理关系

容器管理关系（Container Management Relationship, CMR）是 CMP 2.0 引入的、功能强大的新特性。自从引入 EJB 1.0 后，开发者就需要创建实体对象之间的关系，但在 CMP 2.0 前开发者需要为每个关系编写大量的代码，从而能够抽取关联实体的主键，并存储成冒充的外键域。开发者需要为简单关系乏味地编写大量代码，而对于具有引用完整性要求的复杂关系，更需要耗费几个小时的时间。自从 CMP 2.0 开始，开发者再也不用手工编写关系了。容器能够管理具有引用完整性要求的一对一、一对多及多对多关系。但开发者使用 CMR 的前提是，只能够在本地接口之间定义 CMR，即不能够定义不同 JVM⁷中实体之间的 CMR。

开发者创建 CMR 需要遵循两个步骤：创建 cmr-field 抽象访问方法和在 ejb-jar.xml 文件中声明关系。下面的两小节内容分别描述这两项主题。

11.5.1 cmr-field 抽象访问方法

cmr-field 抽象访问方法除了单值关系必须返回关联实体的本地接口、多值关系只能返回 java.util.Collection（或 java.util.Set）对象外，其他方面同 cmp-field 方法语义（signature）相同。同 cmp-field 一样，处于关系中的两个实体必须有一个提供了 cmr-field 抽象访问方法⁷。比如，声明 Organization 和 Gangster 之间的一对多关系，首先开发者需要往 OrganizationBean 类添加如列表 11-15 所示的代码：

列表 11-15 集合值 cmr-field 抽象访问方法声明

```
public abstract class OrganizationBean implements EntityBean {
    public abstract Set getMemberGangsters();
    public abstract void setMemberGangsters(Set gangsters);
}
```

其次，往 GangsterBean 类添加如列表 11-16 所示的代码：

列表 11-16 单值 cmr-field 抽象访问方法声明

```
public abstract class GangsterBean implements EntityBean {
    public abstract Organization getOrganization();
    public abstract void setOrganization(Organization org);
}
```

尽管列表 11-15 和列表 11-16 中每个实体 Bean 都声明了 cmr-field，但实际上只需要关系中的一个实体必须提供一套访问方法。同 cmp-field 一样，cmr-field 要求同时提供 getter 和 setter 访问方法。

⁷ EJB 规范甚至不允许创建同一 JVM 中不同应用间实体的关系。

11.5.2 关系声明

在 ejb-jar.xml 文件中声明关系是很复杂的操作，并且容易出错。使用 XML 声明关系就如同 Visual Basic 语法一样，很容易导致不一致。配置关系的最好办法是借助于工具，比如 XDoclet，或者参考现有的、使用中的关系。列表 11-17 给出了 Organization-Gangster 关系声明。

列表 11-17 ejb-jar.xml 关系声明

```
<ejb-jar>
  <relationships>
    <ejb-relation>
      <ejb-relation-name>Organization-Gangster</ejb-relation-name>
      <ejb-relationship-role>
        <ejb-relationship-role-name>org-has-gangsters
        </ejb-relationship-role-name>

        <multiplicity>One</multiplicity>

        <relationship-role-source>
          <ejb-name>OrganizationEJB</ejb-name>
        </relationship-role-source>

        <cmr-field>
          <cmr-field-name>memberGangsters</cmr-field-name>
          <cmr-field-type>java.util.Set</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>

      <ejb-relationship-role>
        <ejb-relationship-role-name>gangster-belongs-to-org
        </ejb-relationship-role-name>

        <multiplicity>Many</multiplicity>
        <cascade-delete/>

        <relationship-role-source>
          <ejb-name>GangsterEJB</ejb-name>
        </relationship-role-source>

        <cmr-field>
          <cmr-field-name>organization</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
    </ejb-relation>
```



```
</relationships>
</ejb-jar>
```

通过列表可以看出，各个关系都是使用顶级 `relationships`⁸ 元素中的 `ejb-relation` 元素声明的。每个 `ejb-relation` 使用两个 `ejb-relationship-role` 元素（关系的各个实体各使用一个）。`ejb-relationship-role` 元素的详细解释如下：

- **ejb-relationship-role-name**：可选元素，用于声明角色，并匹配 `jbosscmp-jdbc.xml` 中的数据库映射。该元素取值不能同关联的角色名相同。
- **multiplicity**：必须元素，取值是 “One”，或 “Many”。请注意，所有的 XML 元素对大小写敏感。
- **cascade-delete**：可选元素。如果开发者指定它，当双亲实体被删除时，JBossCMP 将删除子实体。该选项仅用于对方关系的 `multiplicity` 是 “One” 的另一方，即子实体。默认情况下，不使用级联删除。
- **relationship-role-source/ejb-name**：必须元素，给出存在角色的实体名。
- **cmr-field/cmr-field-name**：如果实体存在 `cmr-field` 抽象访问方法，则开发者使用该元素指定实体 `cmr-field` 名。
- **cmr-field/cmr-field-type**：`cmr-field` 的类型。其值必须是 `java.util.Collection`，或者 `java.util.Set`。该元素仅仅用于 `cmr-field` 抽象访问方法是集合值的情形。

在添加 `cmr-field` 抽象访问方法和声明关系后，关系应该可以起作用了。更多有关关系的资料，请参考 EJB 2.0 规范的 10.3 节。下节内容将讨论关系的数据库映射。

11.5.3 关系映射

用户可以使用外键或单独的关系表映射关系。默认情况下，一对一和一对多关系使用外键映射风格，而多对多关系只使用关系表映射风格。关系映射是借助于 `jbosscmp-jdbc.xml` 描述符中 `relationship` 的 `ejb-relation` 元素声明的。而关系又是由 `ejb-jar.xml` 文件中的 `ejb-relation-name` 标识的。图 11-7 给出了 `jbosscmp-jdbc.xml` 中 `ejb-relation` 元素的内容模型。

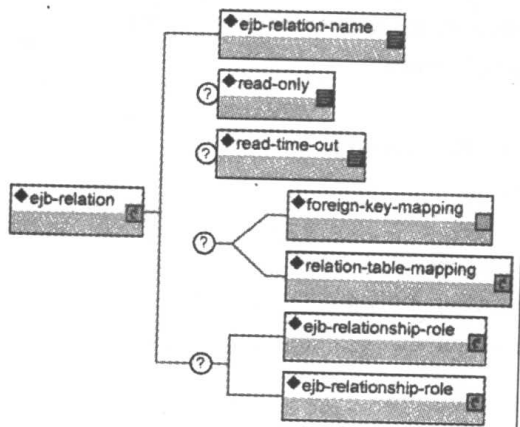


图 11-7 jbosscmp-jdbc.xml ejb-relation 内容模型

⁸这是规范引入不一致性的初始位置。如果规范使用如下标签：`relationships`、`relationship` 及 `relationship-name`，则更容易使用。

用于 Organization-Gangster 关系映射声明的基本模板如列表 11-18 所示。

列表 11-18 jbosscomp-jdbc.xml 关系映射模板

```
<jbosscomp-jdbc>
  <relationships>
    <ejb-relation>
      <ejb-relation-name>Organization-Gangster</ejb-relation-name>
      <foreign-key-mapping/>

      <ejb-relationship-role>
        <ejb-relationship-role-name>org-has-gangsters
      </ejb-relationship-role-name>
      <key-fields>
        <key-field>
          <field-name>name</field-name>
          <column-name>organization</column-name>
        </key-field>
      </key-fields>
    </ejb-relationship-role>

    <ejb-relationship-role>
      <ejb-relationship-role-name>gangster-belongs-to-org
    </ejb-relationship-role-name>
    <key-fields/>
  </ejb-relationship-role>
</ejb-relation>
</relationships>
</jbosscomp-jdbc>
```

在声明关系映射的 `ejb-relation-name` 后，开发者可以使用 `foreign-key-mapping` 元素或 `relation-table-mapping` 元素声明映射风格，这两个元素将在下面的内容中讨论。这里的 `read-only` 和 `read-time-out` 元素同 `entity` 元素中的语义相同。其中，`ejb-relationship-role` 元素是可选元素，但如果声明了一个，则另一个也必须给出。`ejb-relationship-role` 元素的详细描述如下：

- **ejb-relation**：必需元素，给出配置关系的名字。该元素值必须匹配 `ejb-jar.xml` 文件中声明的关系名。
- **read-only**：可选元素。如果为 `true`，则实体 Bean 供应商不能够改变关系值。`read-only` 关系不能够存储到或插入到数据库中。如果某 `setter` 访问方法调用了 `read-only` 关系，则抛出 `EJBException`。
- **read-time-out**：可选元素，给出读取 `read-only` 关系的有效时间（单位：毫秒）。如果为 0，则表示每次事务开始时总是要重新装载该值。如果为 -1，则该值从不超时。如果 `read-only` 为 `false`，则忽略该值。

`ejb-relation` 元素必须含有 `foreign-key-mapping` 元素或 `relation-table-mapping` 元素。它有可能还含有 `ejb-relationship-role` 元素对。

1. 关系角色映射

每个 `ejb-relationship-role` 元素都含有关系中实体 Bean 的具体映射信息，图 11-8 给出了 `ejb-relationship-role` 元素的内容模型。

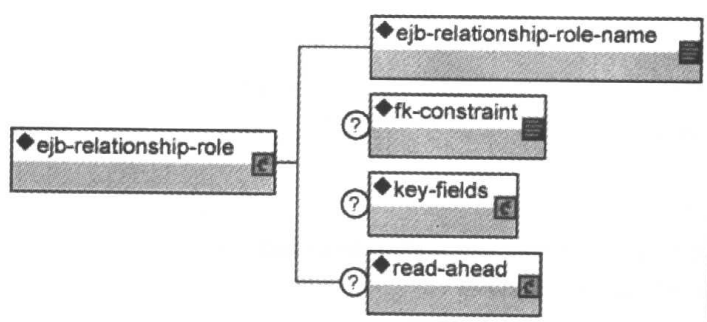


图 11-8 jboss-cmp-jdbc.xml `ejb-relationship-role` 元素的内容模型

其主要内容如下：

- **ejb-relationship-role-name**：必须元素，给出待配置的角色名。该元素值必须匹配 `ejb-jar.xml` 文件中供查询使用的角色名。
- **fk-constraint**：可选元素。如果为 `true`，则表明 JBossCMP 应该添加外键约束给表。只有主表和从表都是由 JBossCMP 在部署期间创建时，JBossCMP 才会添加这种约束。
- **key-fields**：可选元素，指定当前实体的主键域映射。只有在需要完成精确（exact）域映射时，才需要该元素。否则，`key-fields` 元素必须⁹为当前实体 Bean 各个主键域提供 `key-field` 元素。具体细节，下文将给出解释。
- **read-ahead**：可选元素，控制关系的缓存。11.7.4 小节中的“4. 关系”将阐述该元素。

正如上文所述，`key-fields` 元素必须为当前实体 Bean 各个主键域提供 `key-field` 元素。`key-field` 元素和 `cmp-field` 元素使用同一语法，但 `key-field` 不支持 `not-null` 选项。关系表的 `key-field` 自动设置为 `not null`，因为 `key-field` 是表的主键。另外，默认情况下，外键域必须允许为空。这主要是因为当前 JBossCMP 实现是在调用 `ejbCreate` 和 `ejbPostCreate` 之间插入行到数据库中的。既然 EJB 规范规定，如果没有执行到 `ejbPostCreate`，则不允许修改关系，因此外键在初始时设置为 `null`。当删除实体 Bean 时，也存在类似的问题。从 JBoss 3.2.2 开始，使用 `jboss.xml` 中的 `insert-after-ejb-post-create` 容器配置标志能够修改这种插入行为。列表 11-19 给出了相应的 `jboss.xml` 片段。

列表 11-19 `jboss.xml` 片段（描述 `insert-after-ejb-post-create`）

```
<jboss>
...
<container-configurations>
  <container-configuration extends="Standard CMP 2.x EntityBean">
```

⁹ 请注意，如果是外键映射，则该元素可以为空，即当前实体 Bean 不存在外键。但是一对多关系的“多方”必须给出该元素，比如 `Organization-Gangster` 实例中的 `Gangster`。


```

<container-name>INSERT after ejbPostCreate Container</containername>
<insert-after-ejb-post-create>true</insert-after-ejb-post-create>
</container-configuration>
</container-configurations>
</jboss>

```

另一种非空外键解决办法是，将外键元素映射到非空 cmp 域中。这时，开发者能够在 ejbCreate 中使用相应的 cmp 域 setter 方法操作外键域。

图 11-9 给出了 key-fields 元素的内容模型。

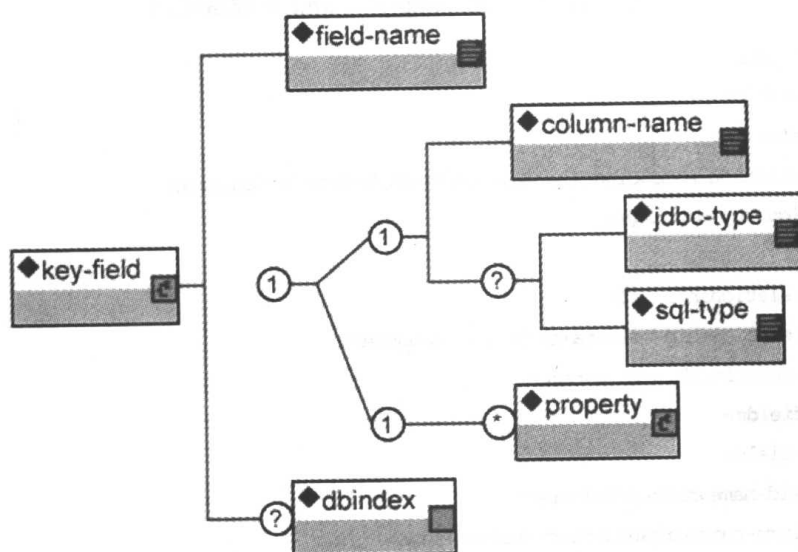


图 11-9 jbosscmp-jdbc key-fields 元素的内容模型

其具体描述如下：

- **field-name**: 必需元素，指定待映射的域。它必须匹配当前实体 Bean 的主键域。
- **column-name**: 开发者使用 column-name 元素指定主键域被存储到的列名。如果关系使用 foreign-key-mapping，则该列将添加到关联实体 Bean 的表中。如果关系使用 relation-table-mapping，则该列将添加到关系表中。column-name 元素不允许使用已映射依赖值类，但是可以使用 property 元素。
 - **jdbc-type**: 当设置 JDBC PreparedStatement 中的参数或从 JDBC ResultSet 装载数据时，开发者才需要使用 JDBC 类型。java.sql.Types 定义了合法类型。
 - **sql-type**: 供创建表语句使用的 SQL 类型。只有数据库厂商才能够决定合法的 SQL 类型。
- **property**: 开发者使用 property 元素定义主键域是依赖值类的映射方法。
- **dbindex**: 可选元素，表明服务器应该为数据库中对应列创建索引，并且索引名是 “<fieldname>_index”。

2. 外键映射

外键映射是用于一对一和一对多关系的最普遍映射风格，但不适合于多对多关系。通过将空 foreign-key-mapping 元素简单地添加给 ejb-relation 元素，即可完成外键映射元素声明。

正如上节所述，如果关系使用 `foreign-key-mapping`，则该列将添加到关联实体 Bean 的表中。如果 `key-fields` 元素为空，则不会创建实体 Bean 的外键。在一对多关系中，必须为多方（Gangster 实例）提供空 `key-fields` 元素，为一方（Organization 实例）提供 `key-fields` 映射。在一对一关系中，允许一个角色或两个角色存在外键。

外键映射不依赖于关系的方向，即在一对一单向关系（只有一方存在访问方法）中，仍然允许一个角色或两个角色存在外键。列表 11-20 给出了 Organization-Gangster 关系的完整外键映射实例，其中加粗部分显示了外键元素。

列表 11-20 jbosscomp-jdbc.xml 外键域映射

```
<jbosscomp-jdbc>
  <relationships>
    <ejb-relation>
      <ejb-relation-name>Organization-Gangster</ejb-relation-name>
      <foreign-key-mapping/>

      <ejb-relationship-role>
        <ejb-relationship-role-name>org-has-gangsters
      </ejb-relationship-role-name>
      <key-fields>
        <key-field>
          <field-name>name</field-name>
          <column-name>organization</column-name>
        </key-field>
      </key-fields>
    </ejb-relationship-role>

    <ejb-relationship-role>
      <ejb-relationship-role-name>gangster-belongs-to-org
    </ejb-relationship-role-name>
    <key-fields/>
  </ejb-relationship-role>
</ejb-relation>
</relationships>
</jbosscomp-jdbc>
```

3. 关系表映射

在操作一对一和一对多关系中，很少使用到关系表映射，它只是多对多关系的惟一映射风格。使用 `relation-table-mapping` 元素能够定义关系表映射，图 11-10 给出了它的内容模型。

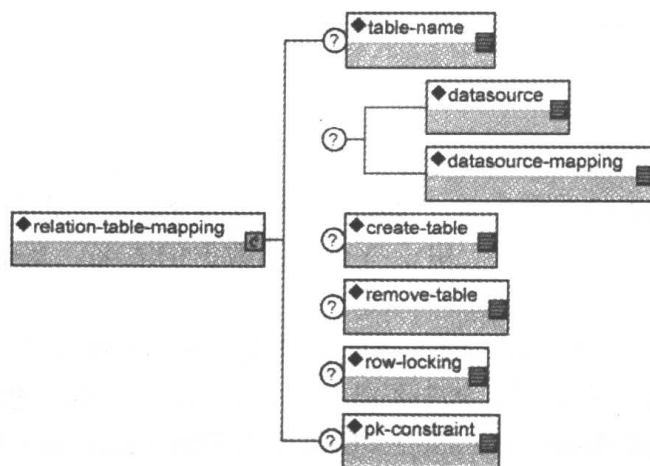


图 11-10 jbossCMP-jdbc.xml relation-table-mapping 元素的内容模型

列表 11-21 给出了 Gangster-Job 关系的 relation-table-mapping 映射实例，其中加粗部分显示了关系表映射元素。

列表 11-21 jbossCMP-jdbc.xml 关系表映射

```

<jbossCMP-jdbc>
  <relationships>
    <ejb-relation>
      <ejb-relation-name>Gangster-Jobs</ejb-relation-name>
      <relation-table-mapping>
        <table-name>gangster_job</table-name>
      </relation-table-mapping>
    </ejb-relation>
    <ejb-relationship-role>
      <ejb-relationship-role-name>gangster-has-jobs
      </ejb-relationship-role-name>
      <key-fields>
        <key-field>
          <field-name>gangsterId</field-name>
          <column-name>gangster</column-name>
        </key-field>
      </key-fields>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <ejb-relationship-role-name>job-has-gangsters
      </ejb-relationship-role-name>
      <key-fields>
        <key-field>
          <field-name>name</field-name>
          <column-name>job</column-name>
        </key-field>
      </key-fields>
    </ejb-relationship-role>
  </relationships>
</jbossCMP-jdbc>
  
```

```
</ejb-relationship-role>
</ejb-relation>
</relationships>
</jboss-cmp-jdbc>
```

relation-table-mapping 元素包含了 entity 元素的部分选项。其中，它的详细描述如下：

- **table-name**: 可选元素，用于持有关系数据的表名。默认值根据实体 Bean 和 cmr-field 名获得。
- **datasource**: 可选元素，给出查找数据源的 jndi-name。所有数据库连接都是通过 datasource 元素获得的。本书不推荐实体 Bean 使用不同的数据源，因为这在很大程度上会约束 finder 和 ejbSelect 所能查询的域（domain）范围。
- **datasource-mapping**: 可选元素，指定使用的 type-mapping 名。
- **create-table**: 可选元素。如果为 true，则 JBoss 将试图为关系创建表。当部署应用时，JBossCMP 在创建表之前会检查数据库表是否存在。如果找到，则记录到日志中，然后不会创建表。在早期开发阶段，由于表结构经常变动，因此在这种场合该元素特别有用。
- **post-table-create**: 可选元素。当创建了数据库表时，立即执行该元素指定的 SQL 语句。但前提是 create-table 为 true，而且该表以前并不存在。
- **remove-table**: 可选元素。如果为 true，则当卸载应用时，JBossCMP 会试图删除关系表。在早期开发阶段，由于表结构经常变动，因此在这种场合该元素特别有用。
- **row-locking**: 可选元素。如果为 true，则 JBossCMP 将锁定事务装入的所有行。当装入实体 Bean 时，大部分数据库都是使用 SELECT FOR UPDATE 语法实现这里的行锁行为的，但是 JBoss 中具体的语法要取决于实体 Bean 使用的 datasource-mapping 中的 row-locking-template。
- **pk-constraint**: 可选元素。如果为 true，则创建表时，JBossCMP 会为表添加主键约束。默认值为 true。

11.6 查 询

CMP 2.0 的另一新的强有力特性是，引入了 EJB 查询语言（EJB Query Language, EJB-QL）和 ejbSelect 方法。在 CMP 1.1 中，不同 EJB 容器使用不同方法指定 finder，这在很大程度上影响了 J2EE 的便携性。在 CMP 2.0 中，EJB-QL 能够以平台无关的方式创建 finder 和 ejbSelect 方法。其中，ejbSelect 方法只是供实体 Bean 实现的私有查询。不同于 finder，即 finder 只能返回同定义它的 home 接口类型相同的实体 Bean，而 ejbSelect 方法能够返回任何实体类型或实体 Bean 的单个域。

EJB-QL 并不是本书的讨论对象，因此这里只是给出基本方法编码和查询声明介绍。更多信息，请开发者参考 Enterprise JavaBeans 规范 Version 2.0 最终发布版的第 11 章，或者有关 CMP 2.0 的优秀文章。

11.6.1 finder 和 ejbSelect 声明

CMP 2.0 并没有修改 finder 声明，还是在实体 Bean 的 home 接口（本地或远程）声明它。定义在本地 home 接口中的 finder 并不会抛出 RemoteException。列表 11-22 给出了 GangsterHome 接口中声明的 findBadDudes_ejbql¹⁰ finder。

列表 11-22 finder 声明

```
public interface GangsterHome extends EJBLocalHome {
    Collection findBadDudes_ejbql(int badness) throws FinderException;
}
```

ejbSelect 方法是在实体 Bean 实现类中声明的，它必须同 cmp-field 和 cmr-field 的抽象访问方法一样，声明为 public abstract。ejbSelect 方法必须声明抛出 FinderException，而不是 RemoteException。列表 11-23 给出了 ejbSelect 方法的实例代码。

列表 11-23 ejbSelect 声明

```
public abstract class GangsterBean implements EntityBean {
    public abstract Set ejbSelectBoss_ejbql(String name)
        throws FinderException;
}
```

11.6.2 EJB-QL 声明

EJB 2.0 规范要求每个 ejbSelect 或 finder 方法（除 findByPrimaryKey）必须在 ejb-jar.xml 文件¹¹中定义相应的 EJB-QL 查询。EJB-QL 查询实际上是包含在 entity 元素中的查询语句。列表 11-24 给出了查询实例。

列表 11-24 ejb-jar.xml 查询声明实例

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      ...
      <query>
        <query-method>
          <method-name>findBadDudes_ejbql</method-name>
          <method-params>
            <method-param>int</method-param>
          </method-params>
        </query-method>
```

¹⁰ 忽略“ejbql”后缀，并不需要它。本文后续内容，将会使用 JBossQL 和 DeclaredSQL 实现该查询，使用后缀的目的只是为了在 jbosscmp-jdbc.xml 文件中隔离不同的查询规范。

¹¹ 目前，JBossCMP 并不对此做强制要求，但后续版本会在部署期间抛出异常以强制该要求。


```
<ejb-ql><![CDATA[
  SELECT OBJECT(g)
  FROM gangster g
  WHERE g.badness > ?1
]]></ejb-ql>
</query>
<query>
  <query-method>
    <method-name>ejbSelectBoss_ejbql</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </query-method>
  <ejb-ql><![CDATA[
    SELECT DISTINCT underling.organization.theBoss
    FROM gangster underling
    WHERE underling.name = ?1 OR underling.nickName = ?1
  ]]></ejb-ql>
</query>
</entity>
</enterprise-beans>
</ejb-jar>
```

EJB-QL 类似于 SQL, 但还是存在一些差别。下面给出 EJB-QL 中一些值得开发者注意的地方:

- EJB-QL 是类型化 (typed) 语言, 即只允许比较同类型 (比如, 字符串只能和字符串进行比较)。
- 在等于表达式中, 变量 (单值路径) 必须位于左边。实例如下¹²:
g.hangout.state = 'CA' 合法
'CA' = g.shippingAddress.state 非法
'CA' = 'CA' 非法
(r.amountPaid * .01) > 300 非法
r.amountPaid > (300 / .01) 合法
- 参数使用的基数从 1 开始, 类似于 java.sql.PreparedStatement。
- 参数只能放在表达式的右边。比如:
gangster.hangout.state = ?1 合法
?1 = gangster.hangout.state 非法

¹² 实例 “(r.amountPaid * .01) > 300” 摘自 Richard Monson-Haefel 的《Enterprise JavaBeans, 3rd》(位于第 244 页), 其目的在于演示 WHERE 从句中算子的使用。本文引用它的目的在于突出显示它是非法 EJB-QL 语法。

11.6.3 覆盖 EJB-QL 到 SQL 的映射

开发者借助于 jbosscmp-jdbc.xml 文件能够覆盖 EJB-QL 到 SQL 的映射。虽然 finder 或 ejbSelect 仍然要求在 ejb-jar.xml 文件中存在 EJB-QL 声明, 但开发者可以不为 ejb-ql 元素提供内容。目前, EJB-QL 覆盖支持的 SQL 类型有 JBossQL、DynamicQL、DeclaredSQL 或 BMP 风格的自定义 finder 方法。所有的上述类型都不是遵循 EJB 2.0 规范进行扩展的, 因此使用它们会影响 EJB 应用的便携性。除了 BMP 自定义 finder 方法之外, 所有的 EJB-QL 覆盖都是用 entity/query 元素进行声明的, 图 11-11 给出了相应的内容模型。

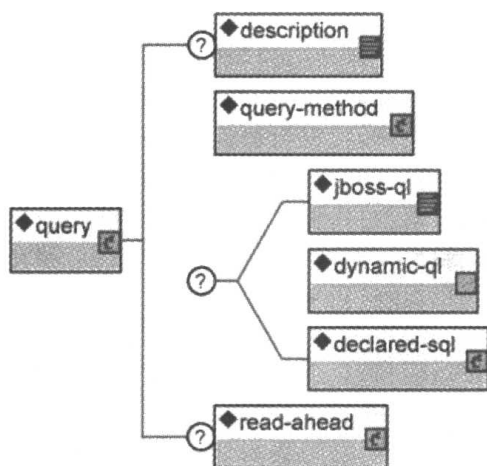


图 11-11 jbosscmp-jdbc.xml query 元素的内容模型

- **description:** 查询的可选性描述。
- **query-method:** 必需元素。指定查询方法。它必须匹配 ejb-jar.xml 文件中 entity 声明的 query-method 方法。
- **jboss-ql、dynamic-ql 及 declared-ql:** 这些元素给出指定查询方式的其他方式。其各自内容, 本章将分节讨论。
- **read-ahead:** 可选元素, 用于优化该查询引用实体 Bean 的其他域装载。在“11.7 优化装载”节中会有其深入的讨论。

11.6.4 JBossQL

JBossQL 是 EJB-QL 的超集, 用于解决 EJB-QL 存在的一些问题。除了提供更加灵活的语法外, 它还增加了新函数、关键字及从句。在写作本书的时候, JBossQL 支持 ORDER BY、OFFSET 及 LIMIT 从句, IN 和 LIKE 操作符中包含参数, 并支持 COUNT、MAX、MIN、AVG、SUM、UCASE 及 LCASE 函数。同时, 在操作查询过程中, 还能在 ejbSelect 方法的 SELECT 从句引入函数。

JBossQL 是通过 jbosscmp-jdbc.xml 文件提供的 query/jboss-ql 元素声明的。该元素含有 JBossQL 查询。列表 11-25 给出了 JBossQL 声明实例, 列表 11-26 给出了相应生成的 SQL。

列表 11-25 jbosscomp-jdbc.xml JBossQL 覆盖

```

<jbosscomp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <query>
        <query-method>
          <method-name>findBadDudes_jbossql</method-name>
          <method-params>
            <method-param>int</method-param>
          </method-params>
        </query-method>
        <jboss-ql><![CDATA[
          SELECT OBJECT(g)
          FROM gangster g
          WHERE g.badness > ?1
          ORDER BY g.badness DESC
        ]]></jboss-ql>
      </query>
    </entity>
  </enterprise-beans>
</ejb-jar>

```

列表 11-26 JBossQL SQL 映射

```

SELECT t0_g.id
FROM gangster t0_g
WHERE t0_g.badness > ?
ORDER BY t0_g.badness DESC

```

JBossQL 的另一项能力是使用 LIMIT 和 OFFSET 函数获得 finder 块状返回结果。比如，为迭代完成含有很多行数据的 jobs 表，开发者可能需要定义如下 findManyJobs_jbossql finder。见列表 11-27。

列表 11-27 使用 LIMIT 和 OFFSET 的 jboss-ql finder 实例

```

<jbosscomp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <query>
        <query-method>
          <method-name>findManyJobs_jbossql</method-name>
          <method-params>
            <method-param>int</method-param>
          </method-params>
        </query-method>
      </query>
    </entity>
  </enterprise-beans>
</ejb-jar>

```

```

        <method-param>int</method-param>
    </method-params>
</query-method>
<jboss-ql><![CDATA[
    SELECT OBJECT(j)
    FROM jobs j
    OFFSET ?1 LIMIT ?2
]]></jboss-ql>
</query>
</entity>
</enterprise-beans>
</ejb-jar>

```

1. DynamicQL

DynamicQL 允许运行时生成和执行 JBossQL 查询。DynamicQL 抽象查询方法将 JBossQL 查询和查询参数作为其方法参数。JBossCMP 编译 JBossQL，并执行生成的 SQL。列表 11-28 给出了 DynamicQL 实例代码。

列表 11-28 DynamicQL 实例代码

```

public abstract class GangsterBean implements EntityBean {
    public Set ejbHomeSelectInStates(Set states)
        throws FinderException
    {
        // generate JBossQL query
        StringBuffer jbossQl = new StringBuffer();
        jbossQl.append("SELECT OBJECT(g) ");
        jbossQl.append("FROM gangster g ");
        jbossQl.append("WHERE g.hangout.state IN (");
        for(int i = 0; i < states.size(); i++)
        {
            if(i > 0)
            {
                jbossQl.append(", ");
            }
            jbossQl.append("?").append(i+1);
        }
        jbossQl.append(") ORDER BY g.name");

        // pack arguments into an Object[]
        Object[] args = states.toArray(new Object[states.size()]);

        // call dynamic-ql query
        return ejbSelectGeneric(jbossQl.toString(), args);
    }
}

```


DynamicQL `ejbSelect` 方法可以定义任何有效的 `ejbSelect` 方法名，但是该方法必须总是带有 `String` 和 `Object` 数组作为参数。开发者使用 `jbosscmp-jdbc.xml` 文件中的 `query/dynamic-ql` 空元素能够声明 DynamicQL。列表 11-29 给出了上述查询的声明。

列表 11-29 `jbosscmp-jdbc.xml` DynamicQL 覆盖实例

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <query>
        <query-method>
          <method-name>ejbSelectGeneric</method-name>
          <method-params>
            <method-param>java.lang.String</method-param>
            <method-param>java.lang.Object []</method-param>
          </method-params>
        </query-method>
        <dynamic-ql/>
      </query>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

2. DeclaredSQL

DeclaredSQL 是基于遗留 JAWS CMP 1.1 引擎 `finder` 声明的，但在 CMP 2.0 中已经进行了再次改进。它主要用于限制 `WHERE` 从句的查询条件，而这些条件 EJB-QL 或 JBossQL 是不具备的。图 11-12 给出了 `declared-sql` 元素的内容模型。

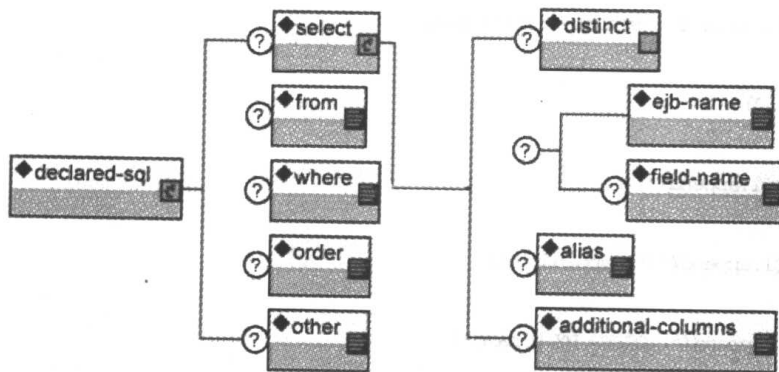


图 11-12 `jbosscmp-jdbc.xml` `declared-sql` 元素的内容模型

- **select:** 指定选择哪些列，其组成元素如下：
 - **distinct:** 如果给出该空元素，JBossCMP 添加 `DISTINCT` 关键字给生成的 `SELECT` 从句。如果方法返回 `java.util.Set`，则默认情况下会使用 `DISTINCT`。
 - **ejb-name:** 待选择的实体 Bean 名。只有在查询 `ejbSelect` 方法使用时才需要它。

- **field-name:** 从指定实体 Bean 选择的 cmp-field 名。默认时, 选择完整的实体 Bean。
- **alias:** 指定主选择表示使用的别名。默认值为 ejb-name 值。
- **additional-columns:** 指定其他待选择的列, 以满足 ejbFinder 中的排序功能, 或辅助 ejbSelect 中的集合函数。
- **from:** 指定追加给生成 SQL 的 from 从句的其他 SQL。
- **where:** 声明用于查询的 where 从句。
- **order:** 指定 order 顺序。
- **other:** 声明追加给 SQL 语句最后的其他 SQL 内容。

列表 11-30 提供了 DeclaredSQL 声明实例, 列表 11-31 给出了相应的生成 SQL。

列表 11-30 jbosscomp-jdbc.xml DeclaredSQL 覆盖

```
<jbosscomp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <query>
        <query-method>
          <method-name>findBadDudes_declaredsql</method-name>
          <method-params>
            <method-param>int</method-param>
          </method-params>
        </query-method>
        <declared-sql>
          <where><![CDATA[ badness > {0} ]]></where>
          <order><![CDATA[ badness DESC ]]></order>
        </declared-sql>
      </query>
    </entity>
  </enterprise-beans>
</ejb-jar>
```

列表 11-31 DeclaredSQL SQL 映射

```
SELECT id
FROM gangster
WHERE badness > ?
ORDER BY badness DESC
```

开发者可以看出, 为选择实体 Bean 的主键, JBossCMP 生成了所需的 SELECT 和 FROM 从句。如果需要, 开发者还可以制定其他 FROM 从句, 以追加到自动生成的 FROM 从句后面。列表 11-32 给出了具有附加 FROM 从句的 DeclaredSQL 声明, 而列表 11-33 给出了其生成的 SQL。

列表 11-32 jbosscomp-jdbc.xml 具有 FROM 从句覆盖的 DeclaredSQL

```

<jbosscomp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      <query>
        <query-method>
          <method-name>ejbSelectBoss_declaredsql</method-name>
          <method-params>
            <method-param>java.lang.String</method-param>
          </method-params>
        </query-method>
        <declared-sql>
          <select>
            <distinct/>
            <ejb-name>GangsterEJB</ejb-name>
            <alias>boss</alias>
          </select>
          <from><![CDATA[, gangster g, organization o]]></from>
          <where><![CDATA[
            (LCASE(g.name) = {0} OR LCASE(g.nick_name) = {0}) AND
            g.organization = o.name AND o.the_boss = boss.id
          ]]></where>
        </declared-sql>
      </query>
    </entity>
  </enterprise-beans>
</ejb-jar>

```

列表 11-33 jbosscomp-jdbc.xml 具有 FROM 从句的 DeclaredSQL SQL 映射

```

SELECT DISTINCT boss.id
FROM gangster boss, gangster g, organization o
WHERE (LCASE(g.name) = ? OR LCASE(g.nick_name) = ?) AND
g.organization = o.name AND o.the_boss = boss.id

```

需要开发者注意的是, FROM 从句最开始使用了逗号。这主要是考虑到容器会将声明的 FROM 从句追加到生成的 FROM 从句后面。另外, FROM 从句也可能以“SQL JOIN”语句开始。既然这是用于 ejbSelect 方法, 则开发者必须提供 select 元素, 以声明待选择实体 Bean。请开发者再次注意, 该查询还声明了一种别名。如果没有声明别名, 则使用 table-name 作为别名, 即结果为, SELECT 从句带有 table_name.field_name 风格的列声明。最后, 请开发者再次注意, 并不是所有的数据库厂商都支持 table_name.field_name 语法, 因此还是推荐使用 alias。列表 11-32 还使用了可选空 distinct 元素, 使得 SELECT 从句使用了 SELECT DISTINCT 声明。ejbSelect 方法也可以使用 DeclaredSQL 声明, 以选择 cmp-field。列表 11-34 给出了 DeclaredSQL 实例, 它选择 Organization 操作的所有邮编,

而列表 11-35 给出了对应的生成 SQL。

列表 11-34 jbosscmp-jdbc.xml DeclaredSQL ejbSelect 覆盖

```
<jbosscmp-jdbc>
<enterprise-beans>
<entity>
<ejb-name>OrganizationEJB</ejb-name>
<query>
<query-method>
<method-name>ejbSelectOperatingZipCodes_declaredsql</methodname>
<method-params>
<method-param>java.lang.String</method-param>
</method-params>
</query-method>
<declared-sql>
<select>
<distinct/>
<ejb-name>LocationEJB</ejb-name>
<field-name>zipCode</field-name>
<alias>hangout</alias>
</select>
<from><![CDATA[ , organization o, gangster g ]]></from>
<where><![CDATA[
    LCASE(o.name) = {0} AND o.name = g.organization AND
    g.hangout = hangout.id
]]></where>
<order><![CDATA[ hangout.zip ]]></order>
</declared-sql>
</query>
</entity>
</enterprise-beans>
</ejb-jar>
```

列表 11-35 jbosscmp-jdbc.xml DeclaredSQL ejbSelect SQL 映射

```
SELECT DISTINCT hangout.zip
FROM location hangout, organization o, gangster g
WHERE LCASE(o.name) = ? AND o.name = g.organization AND g.hangout =
hangout.id
ORDER BY hangout.zip
```

JBossCMP DeclaredSQL 使用了全新的参数处理系统，它支持实体 Bean 和 DVC 参数。参数使用了波形括号“{}”括起它，并且它使用的基数从 0 开始，同 EJB-QL 参数使用的基数 1 不同。JBossCMP DeclaredSQL 一共存在 3 种参数分类：简单、DVC 及实体 Bean。

- 简单参数：除了已知（映射）DVC 或实体 Bean 之外，其他任何类型的参数都可以指定为简单参数。简单参数只含有参数序号，比如{0}。当设置了简单参数时，

entity 元素的 `datasource-mapping` 值内容决定了设置该参数的 JDBC 类型。首先序列化未知 DVC，然后将它设置为参数。请开发者注意，大部分数据库都不支持在 WHERE 从句中使用 BLOB 值。

- DVC 参数：已知（映射）DVC。必须将 DVC 参数分解为简单属性（即不再是其他的 DVC）。比如，如果某属性类型为 `ContactInfo`（在 11.4.6 “依赖值类” 小节中有阐述），则有效的参数声明应该是 `{0.email}` 和 `{0.cell.areaCode}`，而不是 `{0.cell}`。这时，决定设置该参数的 JDBC 类型则是由属性的类类型（class type）和 entity 元素的 `datasource-mapping` 共同决定的。其中，用于设置参数的 JDBC 类型是 `dependent-value-class` 元素中声明的 JDBC 类型。
- 实体 Bean 参数：应用中的任何实体 Bean。实体 Bean 参数必须分解成简单主键域或者 DVC 主键域类型的简单属性。比如，如果某参数为 `Gangster` 类型，则有效的参数声明是 `{0.gangsterId}`。如果某实体 Bean 的主键域是 `ContactInfo` 类型，则有效的参数声明是 `{0.info.cell.areaCode}`。只有是实体 Bean 的主键域成员才可以分解（后续版本会删除该限制）。其中，用于设置参数的 JDBC 类型是实体中为相应域声明的 JDBC 类型。

3. BMP 自定义 finder

JBossCMP 延续了 JAWS 支持 Bean 管理持久化自定义 finder 的能力。如果某自定义 finder 匹配到 home 或本地 home 接口中声明的 finder 方法，则 JBossCMP 将总是使用该自定义 finder，而不是在 `ejb-jar.xml` 或 `jbossCMP-jdbc.xml` 文件中声明的其他任何实现。列表 11-36 给出了通过主键集合¹³查找实体 Bean 的简单实例。

列表 11-36 自定义 finder 实例代码

```
public abstract class GangsterBean implements EntityBean {
    public Collection ejbFindByPrimaryKeys(Collection keys)
    {
        return keys;
    }
}
```

11.7 优化装载

优化装载（Optimized Loading）的目标在于，以最少的查询次数，装载事务所要求的最小数量数据。JBossCMP 调优要求开发者掌握丰富的装载过程知识。本节描述 JBossCMP 装载过程的内部细节及其配置。为调优装载过程，开发者需要彻底地理解装载系统，因此

¹³ 这是非常有用的 finder，因为它能够快速地将主键转化为真实实体 Bean 对象，而不用与数据库交互。但它存在一个缺点，即有可能其创建的实体 Bean 的主键在数据库中不存在。如果任何方法调用了这个无效实体 Bean，则抛出 `NoSuchEntityException`。另一个缺点是，返回的实体 Bean 违反了 EJB 规范，因为它实现了 finder。如果不把 JBoss EJB 验证器的 `StrictVerifier` 属性设置为 `false`，则在部署这种实体 Bean 时将导致失败。

本节内容，开发者至少需要阅读两遍。

11.7.1 装载场景

研究装载过程的最容易方式是，浏览其在实际场景的使用。最常见的场景是定位实体 Bean 集合，然后迭代它，并完成一定的操作。列表 11-37 给出了生成含有所有 gangster 的 HTML 表实例。

列表 11-37 装载场景实例代码

```

1 public String createGangsterHtmlTable_none() throws FinderException {
2     StringBuffer table = new StringBuffer();
3     table.append("<table>");
4
5     Collection gangsters = gangsterHome.findAll_none();
6     for(Iterator iter = gangsters.iterator(); iter.hasNext(); )
7     {
8         Gangster gangster = (Gangster)iter.next();
9         table.append("<tr>");
10        table.append("<td>").append(gangster.getName());
11        table.append("</td>");
12        table.append("<td>").append(gangster.getNickName());
13        table.append("</td>");
14        table.append("<td>").append(gangster.getBadness());
15        table.append("</td>");
16        table.append("</tr>");
17    }
18    return table.toString();
19 }

```

假定：这段代码是在单一事务中调用的，并且所有的优化装载都没有被启用。第 5 行，JBossCMP 将执行列表 11-38 所示的查询：

列表 11-38 未优化 findAll 查询

```

SELECT t0_g.id
FROM gangster t0_g
ORDER BY t0_g.id ASC

```

然后，第 8 行，为了装载实例数据库中的 8 个 gangster，JBossCMP 执行如列表 11-39 所示的 8 次查询：

列表 11-39 未优化装载查询

```

SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=0)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=1)

```

```

SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=2)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=3)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=4)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=5)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=6)
SELECT name, nick_name, badness, hangout, organization
FROM gangster WHERE (id=7)

```

该场景存在两个问题。其一，执行了过多的查询次数，因为 JBossCMP 为 findAll 和各个找到的元素各执行了一次查询。人们常称为“n+1”问题¹⁴，下节内容描述的 read-ahead 策略将解决这个问题。其二，由于 JBossCMP 装入了 hangout 和 organization 域¹⁵，但应用从未使用它们，因此装入了未使用域值。在 11.7.4 小节的“2. eager 装载过程”中阐述了 eager 装载配置。表 11-1 显示了查询的执行结果。

表 11-1 执行未优化查询

id	name	nick_name	badness	hangout	organization
0	Yojimbo	Bodyguard	7	0	Yakuza
1	Yakuza	Murder	8	0	Yakuza
2	Yuriko	Four finger	4	2	Yakuza
3	Chow	Killer	9	3	Triads
4	Shogi	Lightning	6	4	Triads
5	Valentino	Pizza-Face	4	5	Mafia
6	Tom	Toothless	2	6	Mafia
7	Corleone	Godfather	6	7	Mafia

11.7.2 装载组

装载系统的配置和优化首先需要在 entity 元素中声明指定的装载组 (load group)。装载组含有 cmp-field 和具有外键的 cmr-field 名，供单个操作装入。列表 11-40 给出了实例配置。

列表 11-40 jbosscmp-jdbc.xml 装载组声明

```
<jbosscmp-jdbc>
```

¹⁴ 原因在于在 JBoss 容器中需要处理返回的查询结果。尽管从表面上看，当执行查询时实际上返回了已选择的实体 Bean。但是，JBoss 仅仅返回了匹配实体 Bean 的主键，除非某方法调用了实体 Bean，否则是不会装载该实体 Bean 的。

¹⁵ 通常情况下，JBossCMP 还会装载 contactInfo 域，但为简化可读性，考虑到 contactInfo 映射为 7 列，因此本实例将它进行失效处理。

```

<enterprise-beans>
  <entity>
    <ejb-name>GangsterEJB</ejb-name>
    ...
    <load-groups>
      <load-group>
        <load-group-name>basic</load-group-name>
        <field-name>name</field-name>
        <field-name>nickName</field-name>
        <field-name>badness</field-name>
      </load-group>
      <load-group>
        <load-group-name>contact info</load-group-name>
        <field-name>nickName</field-name>
        <field-name>contactInfo</field-name>
        <field-name>hangout</field-name>
      </load-group>
    </load-groups>
  </entity>
</enterprise-beans>
</jboss-cmp-jdbc>

```

上述列表声明了两个装载组：basic 和 contact info。请开发者注意，装载组不需要相互排它性。比如，这两个装载组都含有 nickName 域。除了声明的装载组外，JBossCMP 还自动添加名为 “*” 的装载组，它含有实体中每个 cmp-field 和具有外键的 cmr-field。

11.7.3 read-ahead

JBossCMP 中的优化装载称为 “read-ahead”。该术语源于 JAWS，指从被装载实体 Bean 读取行和其接下来若干行的技术，因此称为 read-ahead。JBossCMP 实现了两个主要策略（on-find 和 on-load），供优化前面提到的装载问题使用。在 read-ahead 期间装载的额外数据并没有和内存中的实体 Bean 对象立即关联在一起，除非实际需要这些数据时，JBossCMP 才会将它们转换成实体 Bean。相反，JBossCMP 将它们存储在预装载缓存区，直到被装入到实体 Bean 或事务结束。接下来开始探讨 read-ahead 策略。

1. on-find

当开发者调用查询时，on-find 策略会读取额外行。如果列表 11-37 中描述的查询场景使用了 on-find 优化装载，JBossCMP 将执行如列表 11-41 所示的查询（第 5 行）：

列表 11-41 on-find 优化 findAll 查询

```

SELECT t0_g.id, t0_g.name, t0_g.nick_name, t0_g.badness
FROM gangster t0_g
ORDER BY t0_g.id ASC

```

然后，位于第 8 行，这些所需数据都已经放置在预装载缓存区中，因此不需要执行多余查询。这种策略对于那些返回少量数据的查询特别有效，但开发者如果试图装入大量的

数据到内存¹⁶中, 则这种方法变得效率低下。表 11-2 给出了查询的执行结果。

表 11-2 执行 on-find 优化查询

id	name	nick_name	badness	hangout	organization
0	Yojimbo	Bodyguard	7	0	Yakuza
1	Takeshi	Master	10	1	Yakuza
2	Yuriko	Four finger	4	2	Yakuza
3	Chow	Killer	9	3	Triads
4	Shogi	Lightning	8	4	Triads
5	Valentino	Pizza-Face	4	5	Mafia
6	Toni	Toothless	2	6	Mafia
7	Corleone	Godfather	6	7	Mafia

用于查询的 read-ahead 策略和 load-group 都是在 query 元素中定义的。如果 query 元素中没有声明 read-ahead 策略, 则将使用 entity 元素或 defaults 元素定义的策略。on-find 配置见列表 11-42。

列表 11-42 jbosscomp-jdbc.xml on-find 声明

```
<jbosscomp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      ...
      <query>
        <query-method>
          <method-name>findAll_onfind</method-name>
          <method-params/>
        </query-method>
        <jboss-ql><![CDATA[
          SELECT OBJECT(g)
          FROM gangster g
          ORDER BY g.gangsterId
        ]]></jboss-ql>
        <read-ahead>
          <strategy>on-find</strategy>
          <page-size>4</page-size>
          <eager-load-group>basic</eager-load-group>
        </read-ahead>
      </query>
    </entity>
  </enterprise-beans>
</jbosscomp-jdbc>
```

¹⁶ JBossCMP 在 read-ahead 缓存实现时, 使用了软引用 (soft reference), 因此数据缓存后不久会立即释放掉。

on-find 策略存在的问题有，它必须装载被选择实体 Bean 的其他数据。通常，在 Web 应用中，只需要将结果中的固定几列显示在 Web 页面。既然预装载数据仅仅在事务范围内有效，并且事务被局限在单个 Web HTTP 点击中，那么大部分预装载数据都没有使用到。下节讨论的 on-load 策略解决了这个问题。

2. on-load

当装载实体 Bean 时，on-load 策略块 (block) 会装载其他若干实体的数据，即请求实体 Bean 和接下来的若干实体 Bean，并根据选择的顺序排序¹⁷。该策略假设的理论基础是，用户将会依次访问 find 或 ejbSelect 的返回结果。当执行查询时，JBossCMP 将存储找到的实体 Bean 顺序到列表缓存区中。随后，当需要装载某实体 Bean 时，JBossCMP 用该列表判断待装入实体块。缓存区中列表数量是通过 entity 的 list-cachemax 元素指定的。该策略也用于如下场合，即在使用 on-find 策略装载数据失败时。列表 11-43 给出了应用 on-load 策略执行列表 11-37 第 5 行的查询。

列表 11-43 on-load (未优化) findAll 查询

```
SELECT t0_g.id
FROM gangster t0_g
ORDER BY t0_g.id ASC
```

比如，如果将 on-load/page-size 设置为 4，JBossCMP 将执行如下 2 次查询，以装载实体 Bean 的 name、nickName 及 badness 域，见列表 11-44。

列表 11-44 on-load 已优化装载查询

```
SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=0) OR (id=1) OR (id=2) OR (id=3)

SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=4) OR (id=5) OR (id=6) OR (id=7)
```

其执行结果见表 11-3。

表 11-3 执行 on-load 已优化查询

id	name	nick_name	badness	hangout	organization
0	Yojiro	Bodyguard	7	0	Yakuza
1	Takeshi	Master	10	1	Yakuza
2	Yuriko	Four finger	4	2	Yakuza
3	Chow	Killer	9	3	Triads
4	Shogi	Lightning	8	4	Triads
5	Valentino	Pizza-Face	4	5	Mafia
6	Toni	Toothless	2	6	Mafia
7	Corleone	Godfather	6	7	Mafia

同 on-find 策略一样，on-load 也是在 read-ahead 元素中声明的。列表 11-45 给出了该

¹⁷ 这是遗留 JAWS 1.1 CMP 引擎中的 read-ahead 技术。

实例的 on-load 配置。

列表 11-45 jbosscomp-jdbc.xml on-load 声明

```
<jbosscomp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      ...
      <query>
        <query-method>
          <method-name>findAll_onload</method-name>
          <method-params/>
        </query-method>
        <jboss-ql><![CDATA[
          SELECT OBJECT(g)
          FROM gangster g
          ORDER BY g.gangsterId
        ]]></jboss-ql>
        <read-ahead>
          <strategy>on-load</strategy>
          <page-size>4</page-size>
          <eager-load-group>basic</eager-load-group>
        </read-ahead>
      </query>
    </entity>
  </enterprise-beans>
</jbosscomp-jdbc>
```

3. none

none 策略实际上是反策略。该策略要求系统退回到默认 lazy 装载代码集中。特别地，它也不 read-ahead 任何数据，或记得已寻找到实体 Bean 的顺序。该策略重现了最开始实例中出现的查询和性能问题。none 策略也是声明在 read-ahead 元素中。如果 read-ahead 元素含有 page-size 或 eager-load-group 元素，则 none 策略将忽略它们。列表 11-46 给出了 none 策略声明。

列表 11-46 jbosscomp-jdbc.xml none 声明

```
<jbosscomp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      ...
      <query>
        <query-method>
          <method-name>findAll_none</method-name>
          <method-params/>
        </query-method>
      </query>
    </entity>
  </enterprise-beans>
</jbosscomp-jdbc>
```

```

</query-method>
<jboss-ql><![CDATA[
    SELECT OBJECT(g)
    FROM gangster g
    ORDER BY g.gangsterId
]]></jboss-ql>
<read-ahead>
    <strategy>none</strategy>
</read-ahead>
</query>
</entity>
</enterprise-beans>
</jboss-cmp-jdbc>

```

11.7.4 装载过程

在上小节中，很多地方提到“当装载实体 Bean 时”，这是本文故意含糊不清，因为为实体 Bean 指定的提交选项和事务的当前状态决定了何时装载实体。接下来，本文开始描述提交选项和装载过程。

1. 提交选项

提交选项（commit-option）是装载过程的关键地方，它控制了实体 Bean 的数据何时过期。JBoss 支持 4 种提交选项：A、B、C 及 D。Enterprise JavaBeans 规范 Version 2.0 最终发布版第 10.5.9 节描述了前 3 个选项，第 4 个选项只是 JBoss 添加的。各个选项的具体描述如下：

- A: JBossCMP 假定只有它自己访问数据库。因此，JBossCMP 能够缓存在事务之间缓存实体 Bean 的当前值，从而获得性能的改善。正因为这个假设，在 JBossCMP 之外不能够修改它管理的数据。比如，在其他程序或直接使用 JDBC（甚至在 JBoss 中）改变数据，使得数据库状态不一致。
- B: JBossCMP 假定存在不止一个数据库用户，但还是保存了事务之间实体 Bean 的上下文信息，这些上下文信息供优化装载实体 Bean 使用。这是默认的提交选项。
- C: 在事务结束时，JBossCMP 会丢弃所有实体 Bean 的上下文信息。
- D: 这是 JBoss 独有的提交选项。它类似于提交选项 A，但是数据的有效时间是一定的。

jboss.xml 文件声明了提交选项，更多信息请开发者参考第 5 章的内容。列表 11-47 给出了实体 Bean 实例应用提交选项 A 的声明。

列表 11-47 jboss.xml 提交选项声明

```

<jboss>
  <container-configurations>
    <container-configuration>

```



```
<container-name>Standard CMP 2.x EntityBean</container-name>
  <commit-option>A</commit-option>
</container-configuration>
</container-configurations>
</jboss>
```

2. eager 装载过程

CMP 2.0 的一项最重要变化是，从使用类的域到通过抽象访问方法声明 `cmp-field`。在 CMP 1.X 中，容器并不知道事务需要哪些域，因此当装载实体 Bean¹⁸时，容器通过 `eager` 装载方式装载各个域。在 CMP 2.x 中，容器实现了抽象访问方法，因此容器知道何时需要域的数据。JBossCMP 能够配置成，当装载实体 Bean 时，以 `eager` 的装载方式装载某些域，然后以 `lazy` 的装载方式装载其余域。

当装载实体 Bean 时，JBossCMP 必须决定需要装载的域。默认时，JBossCMP 使用选择了该实体 Bean 的最近查询中的 `eager-load-group`。如果该实体 Bean 没有被查询选中，或者最近查询使用了 `none read-ahead` 策略，则 JBossCMP 使用实体声明的默认 `eager-load-group`。列表 11-48 给出了 `eager` 装载配置实例，其将 `most` 装载组设置为 `GangsterEJB` 实体 Bean 的默认 `eager-load-group`。

列表 11-48 jbosscmp-jdbc.xml eager 装载声明

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      ...
      <load-groups>
        <load-group>
          <load-group-name>most</load-group-name>
          <field-name>name</field-name>
          <field-name>nickName</field-name>
          <field-name>badness</field-name>
          <field-name>hangout</field-name>
          <field-name>organization</field-name>
        </load-group>
      </load-groups>
      <eager-load-group>most</eager-load-group>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

如果在事务中初次调用了某实体 Bean，则触发了 `eager` 装载过程。`eager` 装载过程的具体步骤如下。

¹⁸ JBossCMP 的后续发布版将能够保存应用提交选项 B 的、事务之间实体 Bean 的当前数据，并使用最近更新乐观锁验证，仍然可以适用当前数据。对于含有大量数据的实体 Bean 而言，这将获得重大的性能改善。

步骤

(1) 如果实体 Bean 上下文仍然有效, 则不需要装载任何数据, 因此装载过程结束。当时使用提交选项 A 或选项 D, 并且数据还未超时, 则实体 Bean 的上下文有效。

(2) 刷新 (flush) 实体 Bean 上下文中的任何剩余数据, 从而保证原有数据不会参与到新的装载过程中。

(3) 将主键值重新取回到主键域中。实际上, 主键对象独立于域, 并且在步骤 (2) 后需要重新装载。

(4) 位于预装载缓存区中用于该实体 Bean 的所有数据装入到域中。

(5) JBossCMP 决定还需装载的其他域。通常, 待装载域是由 entity 的 eager-load-group 决定的, 但如果实体是通过具有 on-find 或 on-load read-ahead 策略的查询和 cmr-field 获得的, 则不是由它决定。如果所有域都已经被装载, 则装载过程跳到步骤 (7)。

(6) 执行查询, 以选择所需列。如果实体 Bean 使用了 on-load 策略, 则将装入 page-size 指定大小的行数据。当前实体 Bean 的数据存储在上下文中, 其他实体 Bean 数据存储在预装载缓存区中。

(7) 调用实体 Bean 的 ejbLoad 方法。

3. lazy 装载过程

lazy 装载构成了 eager 装载的另一半。如果某域不是 eager 装载, 则它必须使用 lazy 装载。当实体 Bean 访问未装载域, JBossCMP 装载该域及与该未装载域同一 lazy-load-group 的其他域。然后, JBossCMP 完成 set join 操作, 并删除那些已装载的域。列表 11-49 给出了配置实例。

列表 11-49 jbossCMP-jdbc.xml lazy 装载组声明

```
<jbossCMP-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>GangsterEJB</ejb-name>
      ...
      <load-groups>
        <load-group>
          <load-group-name>basic</load-group-name>
          <field-name>name</field-name>
          <field-name>nickName</field-name>
          <field-name>badness</field-name>
        </load-group>
        <load-group>
          <load-group-name>contact info</load-group-name>
          <field-name>nickName</field-name>
          <field-name>contactInfo</field-name>
          <field-name>hangout</field-name>
        </load-group>
      </load-groups>
    </entity>
  </enterprise-beans>
</jbossCMP-jdbc>
```

```
</load-groups>
...
<lazy-load-groups>
  <load-group-name>basic</load-group-name>
  <load-group-name>contact info</load-group-name>
</lazy-load-groups>
</entity>
</enterprise-beans>
</jboss-cmp-jdbc>
```

当实体 Bean 供应商使用该配置调用 `getName()` 时, JBossCMP 装入 `name`、`nickName` 及 `badness` (假设它们都还未装载)。当实体 Bean 供应商使用该配置调用 `getNickName()` 时, JBossCMP 装入 `name`、`nickName`、`badness`、`contactInfo` 及 `hangout`。lazy 装载过程的具体步骤如下。

步骤

- (1) 将预装载缓存区中该实体 Bean 的所有数据装载到域中。
- (2) 如果预装载缓存区中含有所需的域值, 则 lazy 装载过程结束。
- (3) JBossCMP 寻找含有该域的所有 lazy-load-group, 然后对这些组进行 set join 操作, 并删除已装载的任何域。
- (4) 执行查询, 以选择所需列。在 basic 装载过程中, JBossCMP 可能装入实体 Bean 块。当前实体 Bean 的数据存储在上下文中, 其他实体 Bean 数据存储在预装载缓存区中。

4. 关系

关系是 lazy 装载中的一种特殊情形, 因为 `cmr-field` 既是域, 又是查询。作为域, 它能用于 on-load 块装载, 即装入当前 (待寻找的) 实体 Bean 和紧接它的若干个实体 Bean 的 `cmr-field` 值。作为查询, 能够使用 on-find 预装载相关实体 Bean 的域值。

同时, 研究装载的最容易方式是浏览其在实际场合的使用。比如, 列表 11-50 给出了实例代码, 其生成含有各个 gangster 和各自 hangout 的 HTML 表。

列表 11-50 关系 lazy 装载实例代码

```
20 public String createGangsterHangoutHtmlTable() throws FinderException
21 {
22     StringBuffer table = new StringBuffer();
23     table.append("<table>");
24
25     Collection gangsters = gangsterHome.findAll_onfind();
26     for(Iterator iter = gangsters.iterator(); iter.hasNext(); )
27     {
28         Gangster gangster = (Gangster)iter.next();
29         Location hangout = gangster.getHangout();
30         table.append("<tr>");
31         table.append("<td>").append(gangster.getName());
```

```

32 table.append("</td>");
33 table.append("<td>").append(gangster.getNickName());
34 table.append("</td>");
35 table.append("<td>").append(gangster.getBadness());
36 table.append("</td>");
37 table.append("<td>").append(hangout.getCity());
38 table.append("</td>");
39 table.append("<td>").append(hangout.getState());
40 table.append("</td>");
41 table.append("<td>").append(hangout.getZipCode());
42 table.append("</td>");
43 table.append("</tr>");
44 }
45 table.append("</table>");
46 return table.toString();
47 }

```

本实例中，Gangster findAll_onfind 配置与 on_find 节讨论的相同。列表 11-51 给出了 Location 实体 Bean 和 Gangster-Hangout 关系的配置。

列表 11-51 jbosscomp-jdbc.xml 关系 lazy 装载配置

```

<jbosscomp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>LocationEJB</ejb-name>
      <load-groups>
        <load-group>
          <load-group-name>quick info</load-group-name>
          <field-name>city</field-name>
          <field-name>state</field-name>
          <field-name>zipCode</field-name>
        </load-group>
      </load-groups>
      <eager-load-group/>
    </entity>
  </enterprise-beans>

  <relationships>
    <ejb-relation>
      <ejb-relation-name>Gangster-Hangout</ejb-relation-name>
      <foreign-key-mapping/>

      <ejb-relationship-role>
        <ejb-relationship-role-name>gangster-has-a-hangout
        </ejb-relationship-role-name>
      <key-fields/>
    </ejb-relation>
  </relationships>

```



```

    <read-ahead>
      <strategy>on-find</strategy>
      <page-size>4</page-size>
      <eager-load-group>quick info</eager-load-group>
    </read-ahead>
  </ejb-relationship-role>

  <ejb-relationship-role>
    <ejb-relationship-role-name>hangout-for-a-gangster
  </ejb-relationship-role-name>
    <key-fields>
      <key-field>
        <field-name>locationID</field-name>
        <column-name>hangout</column-name>
      </key-field>
    </key-fields>
  </ejb-relationship-role>
</ejb-relation>
</relationships>
</jbosscomp-jdbc>

```

在列表 11-50 的第 25 行，JBossCMP 将执行如下查询，见列表 11-52。

列表 11-52 on-find 优化 findAll 查询

```

SELECT t0_g.id, t0_g.name, t0_g.nick_name, t0_g.badness
FROM gangster t0_g
ORDER BY t0_g.id ASC

```

在列表 11-50 的第 29 行，JBossCMP 执行下列两个查询，以装载 hideout 的 city、state 及 zip 域。见列表 11-53。

列表 11-53 on-find 优化关系装载查询

```

SELECT gangster.id, gangster.hangout,
location.city, location.st, location.zip
FROM gangster, location
WHERE (gangster.hangout=location.id) AND
((gangster.id=0) OR (gangster.id=1) OR
(gangster.id=2) OR (gangster.id=3))
SELECT gangster.id, gangster.hangout,
location.city, location.st, location.zip
FROM gangster, location
WHERE (gangster.hangout=location.id) AND
((gangster.id=4) OR (gangster.id=5) OR
(gangster.id=6) OR (gangster.id=7))

```

表 11-4 给出了执行的结果。

表 11-4 on-find 优化关系查询执行

Gangster					Location			
id	name	nick_name	badness	hangout	id	city	st	zip
0	Yojimbo	Bodyguard	7	0	0	San Fran	CA	94108
1	Takeshi	Master	10	1	1	San Fran	CA	94133
2	Yunko	Four finger	4	2	2	San Fran	CA	94133
3	Chow	Killer	9	3	3	San Fran	CA	94133
4	Shogi	Lightning	8	4	4	San Fran	CA	94133
5	Valentino	Pizza Face	4	5	5	New York	NY	10017
6	Toni	Toothless	2	6	6	Chicago	IL	60661
7	Corleone	Godfather	6	7	7	Las Vegas	NV	89109

11.7.5 事务

本章阐述的实例都运行在事务中。事务粒度是优化装载中最主要的因素，因为事务定义了预装载数据的生命周期。如果事务完成，则提交或回滚，使得预装载缓存区中的数据会全部丢失。因此，它将导致一些严重的、负面的性能影响。

运行没有事务的应用，开发者能够看出事务对性能的影响很严重，比如列表 11-54 给出的实例类似于列表 11-37。该实例使用 on-find 优化查询，并选择前 4 个 gangster（为保持返回集合最小），而且它没有处于事务中。

列表 11-54 无事务装载实例代码

```

48 public String createGangsterHtmlTable_no_tx() throws FinderException
49 {
50     StringBuffer table = new StringBuffer();
51     table.append("<table>");
52
53     Collection gangsters = gangsterHome.findFour();
54     for(Iterator iter = gangsters.iterator(); iter.hasNext(); )
55     {
56         Gangster gangster = (Gangster)iter.next();
57         table.append("<tr>");
58         table.append("<td>").append(gangster.getName());
59         table.append("</td>");
60         table.append("<td>").append(gangster.getNickName());
61         table.append("</td>");
62         table.append("<td>").append(gangster.getBadness());
63         table.append("</td>");
64         table.append("</tr>");
65     }
66
67     table.append("</table>");

```

```

68 return table.toString();
69 }

```

第 53 行, 执行查询的结果如列表 11-55 所示。

列表 11-55 无事务 on-find 优化 findAll 查询

```

SELECT t0_g.id, t0_g.name, t0_g.nick_name, t0_g.badness
FROM gangster t0_g
WHERE t0_g.id < 4
ORDER BY t0_g.id ASC

```

通常, 只会执行惟一一次查询。但既然代码不是运行在事务中, 则只要 `findAll` 返回, 所有预装载数据都将丢弃。然后, 第 56 行, JBossCMP 将执行如下 4 个查询 (每个循环 1 次)¹⁹, 见列表 11-56。

列表 11-56 无事务 on-load 优化装载查询

```

SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=0) OR (id=1) OR (id=2) OR (id=3)
SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=1) OR (id=2) OR (id=3)
SELECT id, name, nick_name, badness
FROM gangster
WHERE (id=2) OR (id=3)
SELECT name, nick_name, badness
FROM gangster
WHERE (id=3)

```

图 11-13 给出了查询的结果。

id§	name§	nick_name§	badness§
	Yojimbo	Bodyguard	7
	Takeshi	Master	10
	Yuriko	Four finger	4
	Chow	Killer	9

图 11-13 无事务 on-find 优化查询执行

因此, 这里的性能比使用 `none read-ahead` 差很多, 因为从数据库装入了过多的数据。下列等式能够计算出装载的行数:

¹⁹ 实际情况比这更遭。JBossCMP 执行各个查询 3 次, 即每次访问 `cmp-field` 都需要执行 1 次。因为, `cmp-field` 访问方法调用之间的预装载值都被丢弃了。

$$n + n - 1 + n - 2 + \dots + 1 = \frac{n(n+1)}{2} = O(n^2)$$

本实例的事务只有在调用实体 Bean 时才存在。接下来，用户会问：“如何将我的代码运行在事务中呢？”答案取决于代码在何处运行。如果它运行在 EJB（会话、实体或消息驱动 Bean）中，则方法必须在集成描述符中将 trans-attribute 元素标识为 Required 或 RequiresNew。如果代码没有运行在 EJB 中，则需要用户事务。列表 11-57 给出了通过用户事务访问方法的代码片段。

列表 11-57 用户事务实例代码

```
public String createGangsterHtmlTable_with_tx()
    throws FinderException
{
    UserTransaction tx = null;
    try
    {
        InitialContext ctx = new InitialContext();
        tx = (UserTransaction) ctx.lookup("UserTransaction");
        tx.begin();

        String table = createGangsterHtmlTable_no_tx();

        if (tx.getStatus() == Status.STATUS_ACTIVE)
        {
            tx.commit();
        }
        return table;
    }
    catch (Exception e)
    {
        try
        {
            {
                if (tx != null) tx.rollback();
            }
            catch (SystemException unused)
            {
                // eat the exception we are exceptioning out anyway
            }
            if (e instanceof FinderException)
            {
                {
                    throw (FinderException) e;
                }
            }
            if (e instanceof RuntimeException)
            {
                {
                    throw (RuntimeException) e;
                }
            }
        }
    }
}
```



```
        throw new EJBException(e);
    }
}
```

11.8 乐 观 锁

实体 Bean 支持乐观锁（Optimistic Locking）特性是 JBoss 3.2 才引入的。乐观锁必须同“Instance Per Transaction CMP 2.x EntityBean”CMP 容器配置一起使用，它在应用服务器级不使用悲观锁（Pessimistic Locking）。然后，扩展容器配置，以使用 `org.jboss.ejb.plugins.lock.JDBCOptimisticLock` 锁，其行为由 `jbosscmp-jdbc.xml` 锁策略驱动。列表 11-58 给出了 `jboss.xml` 描述符实例，其阐述了配置原型。

列表 11-58 乐观锁实体容器的原型配置

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss PUBLIC
    "-//JBoss//DTD JBOSS 3.2//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_3_2.dtd">
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>EntityGroupLocking</ejb-name>
      <local-jndi-name>local/EntityGroupLocking</local-jndi-name>
      <configuration-name>Optimistic Container</configuration-name>
    </entity>
  </enterprise-beans>
  ...
  <container-configurations>
    <container-configuration
      extends="Instance Per Transaction CMP 2.x EntityBean">
      <container-name>Optimistic Container</container-name>
      <!-- overrides: org.jboss.ejb.plugins.lock.NoLock -->
      <locking-policy>org.jboss.ejb.plugins.lock.JDBCOptimisticLock
      </locking-policy>
    </container-configuration>
  </container-configurations>
</jboss>
```

该乐观锁配置允许同一实体 Bean 的多个实例同时激活使用。驱动 `JDBCOptimisticLock` 行为的乐观锁策略选择必须保证它们的一致性。乐观锁策略选择定义了一套域，供提交阶段写入已修改数据到数据库中使用。乐观一致性检查维护了，选择域集合的取值同当前事务开始时存在于数据库中的相同取值。通过 `UPDATE WHERE...` 语句能够保证上述的一致性。

开发者使用 `jbosscmp-jdbc.xml` 描述符中的 `entity/optimistic-locking` 元素能够指定乐观

锁策略选择。图 11-14 给出了 optimistic-locking 元素的内容模型，并且其包含的各个元素的具体描述如下。

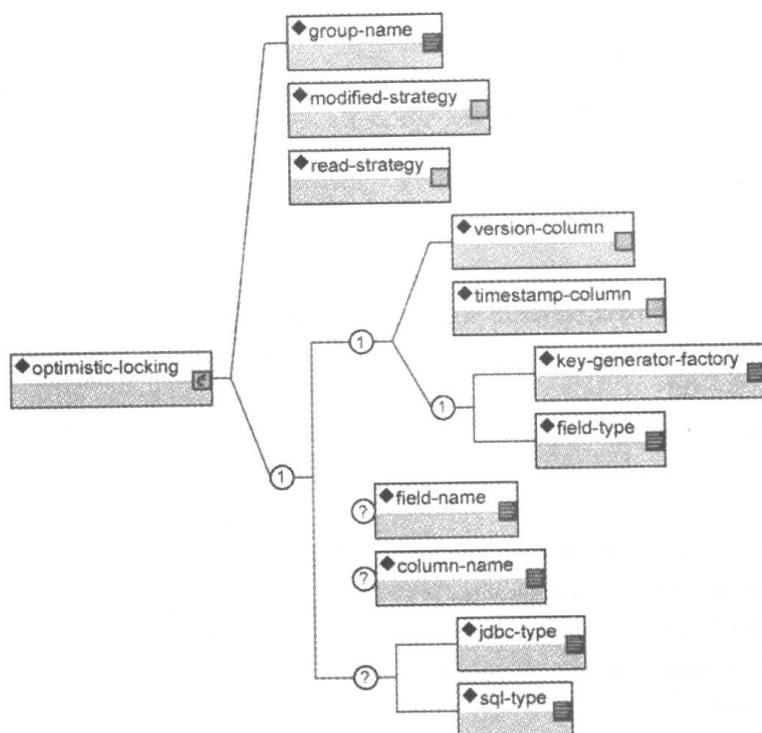


图 11-14 jboss-cmp-jdbc.xml optimistic-locking 元素的内容模型

- **group-name:** group-name 元素指定乐观锁基于装载组的域成员。其值必须匹配实体 Bean 的 load-group-name。匹配的组成员将用于乐观锁。
- **modified-strategy:** 该元素指定乐观锁基于的修改域。该策略表明，在事务期间修改的域将用于乐观锁。
- **read-strategy:** 该元素指定乐观锁基于的读取域。该策略表明，在事务期间被读取或变更的域将用于乐观锁。
- **version-column:** 该元素指定乐观锁基于的版本列策略。如果指定该元素，则将添加 *java.lang.Long* 类型的版本域到实体 Bean，供乐观锁使用。每次更新实体 Bean 都将增加该域的取值。其中，field-name 元素用于 cmp 域名，column-name 用于对应表列。
- **timestamp-column:** 该元素指定乐观锁基于的时间戳列策略。如果指定该元素，则将添加 *java.util.Date* 类型的版本域到实体 Bean，供乐观锁使用。每次更新实体 Bean 都将该列取值设置为当前时间。其中，field-name 元素用于 cmp 域名，column-name 用于对应表列。
- **key-generator-factory:** 该元素指定乐观锁基于键生成。key-generator-factory 元素值是 *org.jboss.ejb.plugins.keygenerator.KeyGeneratorFactory* 实现的 JNDI 名。如果指定该元素，则将添加另一版本域到实体 Bean，供乐观锁使用。每次更新实体 Bean 都将更新键域，具体取值通过键生成器获得。其中，field-name 元素用于 cmp 域名，column-name 用于对应表列。

列表 11-59 给出了 jbosscomp-jdbc.xml 描述符实例，其阐述了所有的乐观锁策略。

列表 11-59 jbosscomp-jdbc.xml 描述符实例（阐述了乐观锁策略）

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jbosscomp-jdbc PUBLIC "-//JBoss//DTD JBOSSCMP-JDBC 3.2//EN" "http://
www.jboss.org/j2ee/dtd/jbosscomp-jdbc_3_2.dtd">

<jbosscomp-jdbc>
  <defaults>
    <datasource>java:/DefaultDS</datasource>
    <datasource-mapping>Hypersonic SQL</datasource-mapping>
  </defaults>

  <enterprise-beans>

    <entity>
      <ejb-name>EntityGroupLocking</ejb-name>
      <create-table>true</create-table>
      <remove-table>true</remove-table>
      <table-name>entitygrouplocking</table-name>
      <cmp-field>
        <field-name>dateField</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>integerField</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>stringField</field-name>
      </cmp-field>

      <load-groups>
        <load-group>
          <load-group-name>string</load-group-name>
          <field-name>stringField</field-name>
        </load-group>
        <load-group>
          <load-group-name>all</load-group-name>
          <field-name>stringField</field-name>
          <field-name>dateField</field-name>
        </load-group>
      </load-groups>

      <optimistic-locking>
        <group-name>string</group-name>
      </optimistic-locking>
    </entity>
  </enterprise-beans>
</jbosscomp-jdbc>
```

```
<entity>
  <ejb-name>EntityModifiedLocking</ejb-name>
  <create-table>true</create-table>
  <remove-table>true</remove-table>
  <table-name>entitymodifiedlocking</table-name>
  <cmp-field>
    <field-name>dateField</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>integerField</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>stringField</field-name>
  </cmp-field>

  <optimistic-locking>
    <modified-strategy/>
  </optimistic-locking>
</entity>

<entity>
  <ejb-name>EntityReadLocking</ejb-name>
  <create-table>true</create-table>
  <remove-table>true</remove-table>
  <table-name>entityreadlocking</table-name>
  <cmp-field>
    <field-name>dateField</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>integerField</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>stringField</field-name>
  </cmp-field>

  <optimistic-locking>
    <read-strategy/>
  </optimistic-locking>
</entity>

<entity>
  <ejb-name>EntityVersionLocking</ejb-name>
  <create-table>true</create-table>
  <remove-table>true</remove-table>
  <table-name>entityversionlocking</table-name>
```



```
<cmp-field>
  <field-name>dateField</field-name>
</cmp-field>
<cmp-field>
  <field-name>integerField</field-name>
</cmp-field>
<cmp-field>
  <field-name>stringField</field-name>
</cmp-field>

<optimistic-locking>
  <version-column/>
  <field-name>versionField</field-name>
  <column-name>ol_version</column-name>
  <jdbc-type>INTEGER</jdbc-type>
  <sql-type>INTEGER(5)</sql-type>
</optimistic-locking>
</entity>

<entity>
  <ejb-name>EntityTimestampLocking</ejb-name>
  <create-table>true</create-table>
  <remove-table>true</remove-table>
  <table-name>entitytimestamplocking</table-name>
  <cmp-field>
    <field-name>dateField</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>integerField</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>stringField</field-name>
  </cmp-field>

  <optimistic-locking>
    <timestamp-column/>
    <field-name>versionField</field-name>
    <column-name>ol_timestamp</column-name>
    <jdbc-type>TIMESTAMP</jdbc-type>
    <sql-type>DATETIME</sql-type>
  </optimistic-locking>
</entity>

<entity>
  <ejb-name>EntityKeyGeneratorLocking</ejb-name>
  <create-table>true</create-table>
```

```

<remove-table>true</remove-table>
<table-name>entitykeygenlocking</table-name>
<cmp-field>
  <field-name>dateField</field-name>
</cmp-field>
<cmp-field>
  <field-name>integerField</field-name>
</cmp-field>
<cmp-field>
  <field-name>stringField</field-name>
</cmp-field>

<optimistic-locking>
  <key-generator-factory>UUIDKeyGeneratorFactory
</key-generator-factory>
  <field-type>java.lang.String</field-type>
  <field-name>uuidField</field-name>
  <column-name>ol_uuid</column-name>
  <jdbc-type>VARCHAR</jdbc-type>
  <sql-type>VARCHAR(32)</sql-type>
</optimistic-locking>
</entity>

</enterprise-beans>

</jboss-cmp-jdbc>

```

11.9 实体命令和主键生成

JBoss 3.2 添加了对主键生成的支持，它位于实体 Bean 类之外。通过自定义实现实体生成命令对象，能够将实体 Bean 插入到持久化存储源中。jboss-cmp-jdbc.xml 描述符的 entity-commands 元素给出了可用命令列表。默认的 entity-command 可以通过 jboss-cmp-jdbc.xml 中的 defaults 元素指定。每个 entity 元素可以指定各自使用的 entity-command 覆盖 defaults 中的 entity-command。图 11-15 给出了 entity-commands 及其子元素的内容模型。

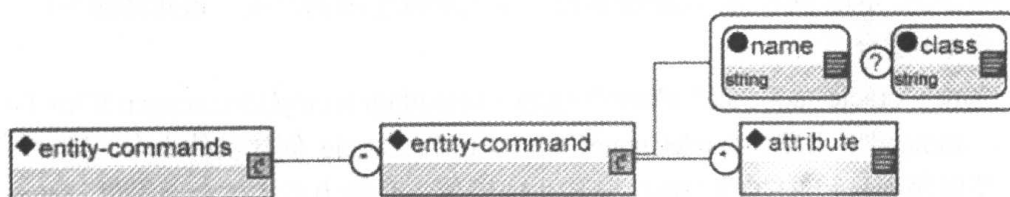


图 11-15 jboss-cmp-jdbc.xml entity-command 元素的内容模型

- **entity-command**: 指定实体生成实现。
 - **entity-command/name**: name 属性指定供 defaults 和 entity 元素引用的 entity-command 名字。
 - **entity-command/class**: entity-command 的 class 属性指定支持键生成的 org.jboss.ejb.plugins.cmp.jdbc.JDBCCreateEntityCommand 实现。如果数据库生成主键，并完成其插入操作，则各数据库厂商的具体命令需要实现 org.jboss.ejb.plugins.cmp.jdbc.JDBCIdentityColumnCreateCommand 类。但如果命令完成生成键的插入操作，则各数据库厂商的具体命令需要实现 org.jboss.ejb.plugins.cmp.jdbc.JDBCInsertPKCreateCommand。
 - **entity-command/attribute**: 可选 attribute 元素，指定 name/value 属性对，供实体命令实现类使用。attribute 元素要求提供 name 属性，指定 name 属性；attribute 元素内容为属性的取值。其中，attribute 值是通过如下方法访问的，即 org.jboss.ejb.plugins.cmp.jdbc.metadata.JDBCEntityCommandMetaData.getAttribute(String)方法。

现存的实体命令

standardjbosscmp-jdbc.xml 存在如下 entity-command 定义：

- name="default" class="org.jboss.ejb.plugins.cmp.jdbc.JDBCCreateEntityCommand", 它是默认的实体生成，standardjbosscmp-jdbc.xml 中 defaults 元素引用了它。JDBCCreateEntityCommand 使用分配给它的主键值执行“INSERT INTO”操作。
- name="no-select-before-insert" class="org.jboss.ejb.plugins.cmp.jdbc.JDBCCreateEntityCommand", 它是“default”的另一实现版本，即在 insert 主键前跳过了 select 操作，通过指向 jboss.jdbc:service=SQL ExceptionProcessor 服务的 name="SQLExceptionProcessor" 属性完成上述步骤。SQLExceptionProcessor 服务提供的 boolean isDuplicateKey(SQLException e)操作能够判断是否侵害了惟一约束条件。
- name="pk-sql" class="org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCPkSqlCreateCommand", JDBCPkSql CreateCommand 执行“INSERT INTO”语句，即通过 pk-sql 属性获得下一个主键值。如果使用它，数据库必须提供序列号（sequence）生成支持。
- name="mysql-get-generated-keys" class="org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCMySQLCreateCommand", JDBCMySQLCreateCommand 使用方法 getGeneratedKeys 从 MySQL 本地 java.sql.Statement 接口实现获取生成的键值。该实体命令仅对 JDK 1.3 或 1.4 适合。
- name="oracle-sequence" class="org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCOracleCreateCommand", JDBCOracleCreateCommand 为 Oracle 创建实体命令，即在单个语句中使用序列号和 RETURNING 从句生成键值。必须为该实体命令提供 sequence 元素以指定序列号列名。
- name="hsqldb-fetch-key" class="org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCHsqldbCreateCommand", JDBCHsqldbCreateCommand 通过执行“CALL IDENTITY()”语句获

得生成的键值后，执行“INSERT INTO”语句。该实体命令仅对 JDK 1.3 或 1.4 适合。

- name="sybase-fetch-key" class="org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCSybaseCreateCommand", JDBCSybaseCreateCommand 通过执行“SELECT @@IDENTITY”语句获得生成的键值后，执行“INSERT INTO”语句。该实体命令仅对 JDK 1.3 或 1.4 适合。
- name="mssql-fetch-key" class="org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCSQLServerCreateCommand", JDBCSQLServerCreateCommand 用于 SQL Server，它使用来自 IDENTITY 列的取值。默认时，使用“SELECT SCOPE_IDENTITY()”，从而减少触发器带来的影响。同时，也可以使用 pk-sql 属性覆盖它，比如 SQL Server Version 7.0。
- name="informix-serial" class="org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCInformixCreateCommand", JDBCInformixCreateCommand 使用 getSerial 方法从 Informix 本地 java.sql.Statement 接口实现获取生成的键值。该实体命令仅对 JDK 1.3 或 1.4 适合。
- name="postgresql-fetch-seq" class="org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCPostgreSQLCreateCommand", JDBCPostgreSQLCreateCommand 用于 PostgreSQL，即获得序列号名。可以使用可选 sequence 属性改变序列号名，其默认值为 \${table}_\${pkColumn}_seq。
- name="key-generator" class="org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCKeyGeneratorCreateCommand", JDBCKeyGeneratorCreateCommand 在从 key-generator-factory 引用的键生成器获得主键值后，执行“INSERT INTO”语句。key-generator-factory 属性必须提供如下实现的 JNDI 绑定名，即 org.jboss.ejb.plugins.keygenerator.KeyGeneratorFactory 类。该实体命令仅对 JDK 1.3 或 1.4 适合。
- name="get-generated-keys" class="org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBC30GeneratedKeysCreateCommand", JDBC30GeneratedKeysCreateCommand 在获得 JDBC3 preparedStatement(String,Statement.RETURN_GENERATED_KEYS)返回的自动生成键值后，执行“INSERT INTO”语句。生成键值的具体方法是 PreparedStatement.getGeneratedKeys。既然要求 JDBC3 支持，则它只适合于提供了支持相应 JDBC 驱动的 JDK 1.4.1 及其以上版本。

列表 11-60 给出了配置实例，其中使用实体命令 hsqldb-fetch-key，并将生成的键值映射到已知主键的 cmp-field 中。

列表 11-60 用于已知主键 cmp-field 的自动生成键配置实例

```
<jboss-cmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>LocationEJB</ejb-name>
      <pk-constraint>false</pk-constraint>
      <table-name>location</table-name>

      <cmp-field>
```



```

    <field-name>locationID</field-name>
    <column-name>id</column-name>
    <auto-increment/>
  </cmp-field>
  ...
  <entity-command name="hsqldb-fetch-key"/>

</entity>
</enterprise-beans>
</jbosscmp-jdbc>

```

列表 11-61 给出了未显式给出 cmp-field 的未知主键的另一实例。

列表 11-61 用于未知主键的自动生成键配置实例

```

<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>LocationEJB</ejb-name>
      <pk-constraint>false</pk-constraint>
      <table-name>location</table-name>

      <unknown-pk>
        <unknown-pk-class>java.lang.Integer</unknown-pk-class>
        <field-name>locationID</field-name>
        <column-name>id</column-name>
        <jdbc-type>INTEGER</jdbc-type>
        <sql-type>INTEGER</sql-type>
        <auto-increment/>
      </unknown-pk>
    </entity>
    ...
    <entity-command name="hsqldb-fetch-key"/>

  </enterprise-beans>
</jbosscmp-jdbc>

```

11.10 defaults

JBoss 3.x 发布版将 JBossCMP 全局 defaults 定义在 server/<servername>/conf 目录的 standardjbosscmp-jdbc.xml 文件中。每个应用都能使用 jbosscmp-jdbc.xml 文件覆盖全局 defaults。默认选项都保存在该配置文件的 defaults 元素中。图 11-16 给出了 defaults 元素的内容模型。

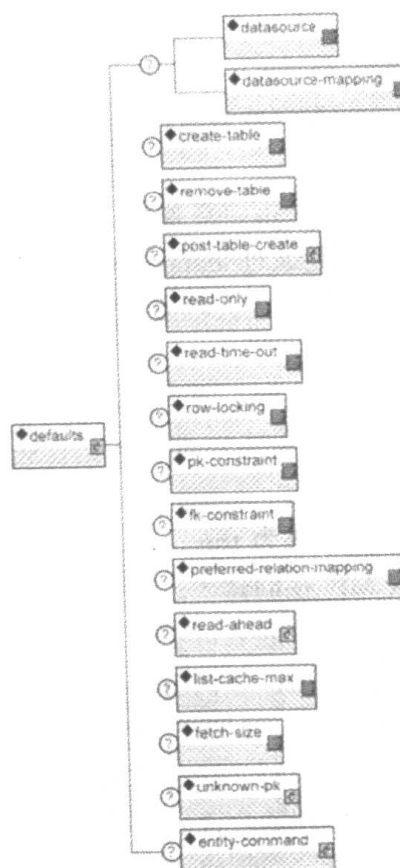


图 11-16 jboss-cmp-jdbc/defaults 的内容模型

列表 11-62 给出了 jboss-cmp-jdbc.xml defaults 声明实例。

列表 11-62 jboss-cmp-jdbc.xml defaults 声明实例

```
<jboss-cmp-jdbc>
  <defaults>
    <datasource>java:/DefaultDS</datasource>
    <datasource-mapping>Hypersonic SQL</datasource-mapping>
    <create-table>true</create-table>
    <remove-table>false</remove-table>
    <read-only>false</read-only>
    <read-time-out>300000</read-time-out>
    <pk-constraint>true</pk-constraint>
    <fk-constraint>false</fk-constraint>
    <row-locking>false</row-locking>
    <preferred-relation-mapping>foreign-key</preferred-relation-mapping>
    <read-ahead>
      <strategy>on-load</strategy>
      <page-size>1000</page-size>
      <eager-load-group>*</eager-load-group>
    </read-ahead>
    <list-cache-max>1000</list-cache-max>
  </defaults>
</jboss-cmp-jdbc>
```

```
</defaults>  
</jboss-cmp-jdbc>
```

各个选项能够应用于实体、关系，或者这两者，并且，在具体实体或关系中覆盖它们。其中，各个选项的详细描述如下：

- **datasource**: 可选元素，用于指定查找数据源的 jndi-name。实体 Bean 或关系表使用的所有数据库连接都是通过 datasource 元素获得的。本书不推荐实体 Bean 使用不同的数据源，因为这在很大程度上会约束 finder 和 ejbSelect 所能查询的域（domain）范围。
- **datasource-mapping**: 可选元素，用于指定 type-mapping 名。type-mapping 元素用于定义具体数据库的 Java 到 SQL 类型映射、SQL 模板及函数映射。在“11.11.2 类型映射”中会讨论类型映射。
- **create-table**: 可选元素。如果为 true，则 JBossCMP 将试图为实体 Bean 创建表。当部署应用时，JBossCMP 在创建表之前会检查数据库表是否存在。如果找到，则记录到日志中，然后不会创建表。在早期开发阶段，由于表结构经常变动，因此在这种场合该元素特别有用。默认值为 false。
- **remove-table**: 可选元素。如果为 true，则 JBossCMP 将试图删除实体 Bean 的表和相应的关系表。当卸载应用时，JBossCMP 会删除表。在早期开发阶段，由于表结构经常变动，因此在这种场合该元素特别有用。默认值为 false。
- **post-table-create**: 可选元素。当创建了数据库表时，立即执行该元素指定的 SQL 语句。但前提是 create-table 为 true，而且该表以前并不存在。
- **read-only**: 可选元素。如果为 true，则实体 Bean 供应商不能改变任何域的值。只读域不允许存储或插入到数据库。如果主键为 read-only，则 create 方法将抛出 CreateException。如果某 read-only 域受到 set 访问方法的调用，则抛出 EJBException。read-only 域对于限定对数据库触发器填入值，比如最近更新的修改很有用。在 cmp-field 级别能够覆盖这里的 read-only 选项。默认值为 false。
- **read-time-out**: 可选元素。读取 read-only 域的有效时间（单位：毫秒）。如果为 0，则表示每次事务开始时总是要重新装载该值。如果为 -1，则该值从不超时。在 cmp-field 级别能够覆盖这里的 read-time-out 选项。如果 read-only 为 false，则忽略该值。默认值为 -1。
- **row-locking**: 可选元素。如果为 true，则 JBossCMP 将锁定事务装入的所有行。当装入实体 Bean 时，大部分数据库都是使用 SELECT FOR UPDATE 语法实现这里的行锁行为的，但是 JBoss 中具体的语法要取决于实体 Bean 使用的 datasource-mapping 中的 row-locking-template。默认值为 false。
- **pk-constraint**: 可选元素。如果为 true，则创建表时，JBossCMP 会为表添加主键约束。默认值为 true。
- **preferred-relation-mapping**: 可选元素，为关系推荐映射风格。其值必须是“foreign-key”或“relation-table”。
- **read-ahead**: 可选元素，用于控制查询结果和实体 Bean 的 cmr-field 的缓存。“11.7.3 read-ahead”小节有深入讨论它。

- **list-cache-max:** 可选元素，用于指定实体 Bean 缓存区中的列表数量。11.7.3 小节中的“2. on-load”有讨论。默认值为 1 000。
- **fetch-size:** 可选元素，指定一次性从底层数据存储源中读入实体 Bean 的数量。默认值为 0。
- **unknown-pk:** 可选元素，定义如何将未知主键类型（java.lang.Object）映射到持久化存储源。
- **entity-command:** 可选元素，定义实体 Bean 生成命令实例。通常，用户使用它定义自定义命令实例，供主键生成使用。“11.9 实体命令和主键生成”节有相关讨论。

11.11 自定义数据源

JBossCMP 为许多数据库包含了预定义的 type-mapping。其中包括：Cloudscape、DB2、DB2/400、Hypersonic SQL、InformixDB、InterBase、MS SQLSERVER、MS SQLSERVER2000、MySQL、Oracle7、Oracle8、Oracle9i、PointBase、PostgreSQL、PostgreSQL 7.2、SapDB、SOLID 及 Sybase。如果用户不喜欢提供的映射，或者没有用户数据库的 type-mapping 映射，则可以定义新的映射。如果用户发现预定义 type-mapping 中存在错误，或者为新数据库创建了新映射，则请考虑以补丁的形式提交给 SourceForge JBoss 项目页。

通过 jbossCMP-jdbc.xml 描述符的 type-mapping 部分能够完成自定义数据库映射。图 11-17 给出了 type-mapping 元素的内容模型。

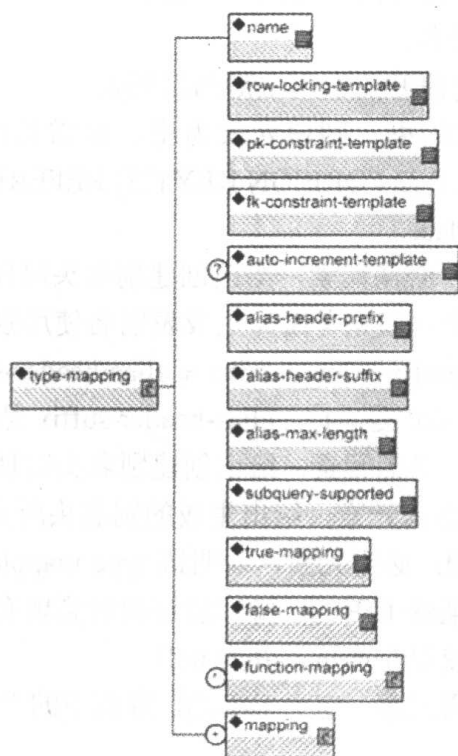


图 11-17 jbossCMP-jdbc.xml type-mapping 的内容模型

各个元素的具体解释如下：

- **name**：必需元素，提供标识数据库自定义名。defaults 和 entity 元素中的 datasource-mapping 需要使用它。
- **row-locking-template**：必需元素，给出 PreparedStatement 模板，供创建已选择行的行锁使用。其中，该模板必须支持如下 3 个参数：
 - 参数 1，select 从句。
 - 参数 2，from 从句，目前还不能保证表的顺序。
 - 参数 3，where 从句。

如果 select 语句不支持行锁，则该元素应该为空。最常见的行锁形式是用于更新的下列语句：SELECT ?1 FROM ?2 WHERE ?3 FOR UPDATE。

- **pk-constraint-template**：必需元素，给出 PreparedStatement 模板，供 create table 语句创建主键约束使用。其中，该模板必须支持如下两个参数：
 - 参数 1，主键约束名，其值总是 pk_{table-name}。
 - 参数 2，用逗号隔开的主键列名列表。

如果 create table 语句不支持主键约束从句，则该元素应该为空。最常见的主键约束形式是：CONSTRAINT ?1 PRIMARY KEY (?2)。

- **fk-constraint-template**：指定模板，供在单独语句中创建外键约束使用。其中，该模板必须支持 5 个参数：
 - 参数 1，表名。
 - 参数 2，外键约束名，其取值总是 fk_{table-name}_{cmr-field-name}。
 - 参数 3，用逗号隔开的外键列名列表。
 - 参数 4，引用表名。
 - 参数 5，用逗号隔开的引用主键列名列表。

如果数据源不支持外键约束，则该元素为空。最常见的外键约束形式是：ALTER TABLE ?1 ADD CONSTRAINT ?2 FOREIGN KEY (?3) REFERENCES ?4 (?5)。

- **auto-increment-template**
- **alias-header-prefix**：必需元素，给出创建别名头时使用的前缀。为防止出现命名冲突，EJB-QL 编译器预先考虑为生成表别名使用别名头。别名头通过如下方式构建：alias-header-prefix + int_counter + alias-header-suffix。比如，别名头 “t0_”，其中 alias-header-prefix 是 “t”，alias-header-suffix 是 “_”。
- **alias-header-suffix**：必需元素，给出创建别名头时使用的后缀。
- **alias-max-length**：必需元素，给出生成的别名头所允许的最大长度。
- **subquery-supported**：必需元素，表明该 type-mapping 的子查询或者为 false，或者为 true。其中，某些 EJB-QL 操作能够映射到现有的子查询中。如果为 false，则 EJB-QL 编译器使用左连接，并为 null。
- **true-mapping**：必需元素，定义 EJB-QL 查询中的 “true” 标识。比如，“TRUE”、“1” 及 “(1=1)”。
- **false-mapping**：必需元素，定义 EJB-QL 查询中的 “false” 标识。比如，“FALSE”、“0” 及 “(1=0)”。

- **function-mapping**: 可选元素, 指定从 EJB-QL 函数到 SQL 实现的一个或多个映射。更多信息, 请参考“11.11.1 函数映射”小节的内容。
- **mapping**: 必需元素, 指定从 Java 类型到对应 JDBC 和 SQL 类型的映射。更多信息, 请参考“11.11.2 类型映射”小节的内容。

11.11.1 函数映射

jbosscmp-jdbc function-mapping 元素的内容模型如图 11-18 所示。

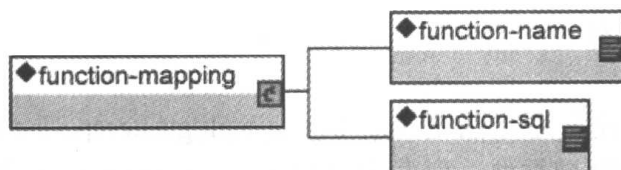


图 11-18 jbosscmp-jdbc function-mapping 元素的内容模型

- **function-name**: 必需元素, 给出 EJB-QL 函数名, 比如 concat 和 substring。
- **function-sql**: 必需元素, 为函数给出适合于底层数据库的 SQL。比如, concat 函数实例有“(?1 || ?2)”、“concat(?1, ?2)”及“(?1 + ?2)”。

11.11.2 类型映射

type-mapping 只不过是 Java 类型和数据库类型之间的简单映射, 而这种类型映射可以使用一套 mapping 元素进行定义, 图 11-19 给出了相应的内容模型。

如果 JBossCMP 不能够找到类型的映射, 它将序列化该对象, 并使用 java.lang.Object 映射。接下来, 给出其 3 个子元素的详细描述:

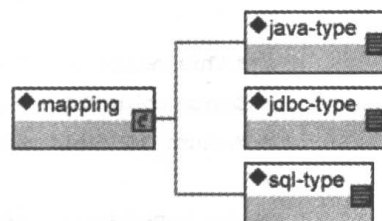


图 11-19 jbosscmp-jdbc mapping 元素的内容模型

- **java-type**: 必需元素, 给出待映射的 Java 类的全限定名。如果该类是原始类型的包裹类, 比如 java.lang.Short, 则还需为原始类型定义映射。
- **jdbc-type**: 必需元素。当设置 JDBC PreparedStatement 中的参数, 或从 JDBC ResultSet 装载数据时, 才需要使用 JDBC 类型。java.sql.Types 定义了合法类型。
- **sql-type**: 必需元素, 供创建表语句使用的 SQL 类型。只有数据库厂商才能够决定合法的 SQL 类型。

列表 11-63 给出了 Oracle9i 中映射元素实例 (short)。

列表 11-63 Oracle9i 的 short 映射实例

```

<jbosscmp-jdbc>
  <type-mappings>
    <type-mapping>
      <name>Oracle9i</name>
    
```

```
...
<mapping>
  <java-type>java.lang.Short</java-type>
  <jdbc-type>NUMERIC</jdbc-type>
  <sql-type>NUMBER(5)</sql-type>
</mapping>
</type-mapping>
</type-mappings>
</jbosscmp-jdbc>
```

11.11.3 用户类型映射

用户类型映射允许用户通过指定 `org.jboss.ejb.plugins.cmp.jdbc.Mapper` 接口的实例将 JDBC 列类型映射到 CMP 域类型。列表 11-64 给出了 Mapper 接口定义。

列表 11-64 `org.jboss.ejb.plugins.cmp.jdbc.Mapper` 接口

```
public interface Mapper
{
    /** This method is called when CMP field is stored.
     * @param fieldValue - CMP field value
     * @return column value.
     */
    Object toColumnValue(Object fieldValue);

    /** This method is called when CMP field is loaded.
     * @param columnValue - loaded column value.
     * @return CMP field value.
     */
    Object toFieldValue(Object columnValue);
}
```

一个常见用例是, 将 `Integer` 类型映射成类型安全的 Java enum 实例。`user-type-mappings` 元素由一个或多个 `user-type-mapping` 元素组成, 其中, 图 11-20 给出了 `user-type-mapping` 的内容模型。

- **java-type**: 映射中, `cmp` 域类型的全限定名。
- **mapped-type**: 映射中, 数据库类型的全限定名。
- **mapper**: `Mapper` 接口实现的全限定名。其中, `Mapper` 接口实现用于处理 `java-type` 和 `mapped-type` 之间的转换。

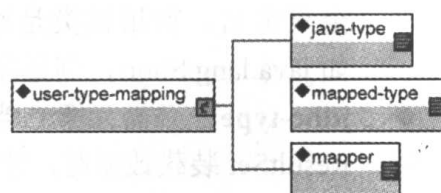


图 11-20 `jbosscmp-jdbc.xml` `user-type-mapping` 元素的内容模型

第 12 章 Web 服务

本章讨论 JBoss 中支持 Web 服务的总体情况。EJB 和 MBean 都能够暴露为 Web 服务，即 Java 和非 Java 客户端能够借助于基于 HTTP 的标准 SOAP 1.1 协议访问它们。其中，JBoss 服务器借助于 Apache Axis 1.1 版本支持 Web 服务。Axis 用于处理 SOAP 细节，而 JBoss 提供自定义部署器和集成。另外，为实现与 XDoclet 的平滑集成，应用服务器还提供了 JBoss.NET 标签库。标准 JBoss.NET Web 服务由无状态会话 Bean 和一套描述 Web 服务属性的元数据组成。

12.1 XDoclet

XDoclet 是很流行的 EJB 开发方式。然而，XDoclet 并不局限于 EJB 开发。XDoclet 方式常称为“面向属性编程 (Attribute Oriented Programming, AOP)”。通常，开发者都将它和“面向方面编程 (Aspect Oriented Programming, AOP)”混淆。然而，尽管它们存在关系，但还是不一样的。面向属性编程标出源代码，从而赋给它更大的含义；面向方面编程应用这种含义到代码中。同时，“方面”通常应用于“属性”，因此它们常携手发展，谁也离不开谁。

在论述本章内容时，本书假定开发者在软件开发中已经在使用 XDoclet。本章也只是扩充了 XDoclet EJB 开发的一点点内容，即介绍 JBoss.NET 的使用。开发者可以通过 <http://xdoclet.sourceforge.net> 获得更多 XDoclet 知识，或者可以阅读由 Craig Walls 和 Norman Richards 写作的、由 Manning 出版社 2003 年出版的《XDoclet in Action》图书。

安装 XDoclet

本书附带的实例存档中含有 XDoclet，XDoclet 的 jar 文件位于 examples/lib 目录中。因此，如果开发者已经有实例存档，则可以跳过本节内容。如果没有，则马上去 SourceForge 下载 XDoclet，并安装它。为获得 XDoclet，请开发者打开网址 <http://xdoclet.sourceforge.net>，然后下载。开发者可以从中选择离自己最近的映像下载。本文这里给出的研究内容都是基于 jboss-3.2.3 源代码发布版的，因此建议开发者使用 jboss-3.2.3 或者其后续版本。

然后，开发者还需要将 JBOSS_DIST/client 中的 xdoclet-module-jboss-net.jar 模块拷贝到 XDoclet 安装的 lib 子目录下。最后，开发者还需要 3.2.1 版本或最新版本的 JBoss 服务器，才能够获得 jboss-net XDoclet 依赖的 J2EE API 库。

12.2 将 Hello World EJB 发布为 Web 服务

本节内容研究将 EJB 发布为 Web 服务的具体步骤。其中，代码位于 examples 存档的 src/main/org/jboss/chap12 目录中。列表 12-1 给出了用 XDoclet 标记完成的 EJB 代码。

列表 12-1 用 XDoclet 标记完成的 HelloWorld EJB 代码

```
/*
 * JBoss, the OpenSource J2EE webOS
 *
 * Distributable under LGPL license.
 * See terms of license at gnu.org.
 */
package org.jboss.chap12.hello;

import javax.ejb.EJBException;

/**
 * The typical Hello Session Bean this time
 * as a web-service.
 * @author jung
 * @version $Revision: 1.1 $
 * @ejb:bean name="Hello"
 *      display-name="Hello World Bean"
 *      type="Stateless"
 *      view-type="remote"
 *      jndi-name="Hello"
 * @ejb:interface remote-class="org.jboss.chap12.hello.Hello"
 * extends="javax.ejb.EJBObject"
 * @ejb:home remote-class="org.jboss.chap12.hello.HelloHome"
 * extends="javax.ejb.EJBHome"
 * @ejb:transaction type="Required"
 */
public class HelloBean
    extends BaseSession implements javax.ejb.SessionBean
{
    /**
     * @ejb:interface-method view-type="remote"
     */
    public String hello(String name)
    {
        return "Hello "+name+"!";
    }
}

/**
```

```

* @ejb:interface-method view-type="remote"
*/
public Object[] complexHello(Object[] query)
{
    Object[] reply = new Object[query.length];
    for(int n = 0; n < query.length; n++)
    {
        HelloObj hello = (HelloObj) query[n];
        System.out.println("hello, "+hello.getMsg());
        reply[n] = new HelloReplyObj(n+": "+hello.getMsg());
    }
    return reply;
}
}

```

其中，BaseSession 只是实现了 EJB 生命周期方法，本书暂不考虑它。假定该会话 Bean 是某应用的必要组成部分，并且能够用于 B2B 环境中为其他客户提供欢迎信息。客户可能使用 Java，也可能不使用 Java。因此，客户就可能不能使用 RMI/IIOP 或 RMI/JRMP 方式直接调用 EJB。

因此，本文决定使用 Web 服务暴露 EJB。同时，本文也打算 EJB 暴露的 Web 服务中存在 hello 和 complexHello 方法。为实现此目的，开发者需要在 javadoc 注释部分为这两个方法标记 @jboss-net:web-method 标签。

```

/**
 * @jboss-net:web-method
 * @ejb:interface-method view-type="remote"
 */
public String hello(String name)
{
    ...
}

/**
 * @jboss-net:web-method
 * @ejb:interface-method view-type="remote"
 */
public Object[] complexHello(Object[] query)
{
    ...
}

```

至此，这些方法暴露为 Web 服务接口的组成部分，但是开发者还需要将 EJB 标记为 Web 服务，因此需要声明 URN。本书建议开发者使用 ejb-name 作为 URN 值。为将 EJB 暴露成 Web 服务，开发者需要添加 @jboss-net:web-service 标签到类级别 javadoc 注释中，并包含 urn=“MyWebServiceName”，以定义 Web 服务的 URN。比如：

```
/**
 *
 * ...
 * @ejb:bean name="Hello"
 *     display-name="Hello World Bean"
 *     type="Stateless"
 *     view-type="remote"
 *     jndi-name="Hello"
 *
 * ...
 * @jboss-net:web-service urn="Hello"
 */
public class HelloBean
```

1. 声明如何处理非简单类型

hello 方法返回 Java 原始类型。不需要其他特殊配置，jboss-net 层就能够处理原始类型。然而，complexHello 方法接受了 HelloObject 类型的 Object[] 数组，并返回 HelloReplyObj 类型的 Object[] 数组。开发者必须把这些类型标记为自定义类型，并且还需声明负责序列化和反序列化这些自定义类型到 XML 的类。为此，开发者需要使用 XDoclet 标签 “@jboss-net.xml-schema”。列表 12-2 给出了自定义类及 jboss-net 标签的使用。

列表 12-2 complexHello 方法使用的非简单类型

```
package org.jboss.examples.ws.hello;

/** A custom data object class that needs to specify a custom serializer
 *
 * @jboss-net.xml-schema urn="hello:HelloObj"
 */

public class HelloObj implements java.io.Serializable
{
    private String msg;

    public HelloObj(String msg)
    {
        this.msg = msg;
    }

    public String getMsg()
    {
        return this.msg;
    }
}

package org.jboss.examples.ws.hello;

/** A custom data object class that needs to specify a custom serializer
 *
```

```

* @jboss-net.xml-schema urn="hello:HelloReplyObj"
*/

public class HelloReplyObj implements java.io.Serializable
{
    private String msg;

    public HelloReplyObj (String msg)
    {
        this.msg = msg;
    }

    public String getMsg()
    {
        return this.msg;
    }
}

```

自定义类型被定义在 urn 参数指定的命名空间下。

2. 编写 Ant build.xml 文件

至此，本文已经完成将 EJB 标记为 JBoss.NET Web 服务的所有内容。为了创建其他文件，比如 home 和远程接口、ejb-jar.xml 及 web-service.xml 描述符，开发者还需创建调用 XDoclet 的 Ant build.xml 文件。首先，需要在 build.xml 中设置一些环境属性，比如源文件位置、构建树结构等。下列 XML 片段摘自 examples 存档中 examples/src/main/org/jboss/chap12/build.xml 文件。因此，在使用基于 XDoclet 的任务之前，开发者需要准备好 XDoclet，即将 XDoclet jar 文件的位置信息告之 Ant。为 XDoclet 创建类路径过程如下：

```

<path id="xdoclet.classpath">
    <pathelement location="${lib.dir}/commons-collections.jar"/>
    <pathelement location="${lib.dir}/xdoclet-module-jboss-net.jar"/>
    <pathelement location="${lib.dir}/xdoclet-jb3.jar"/>
    <pathelement location="${lib.dir}/xdoclet-xjavadoc-jb3.jar"/>
    <pathelement location="${lib.dir}/xdoclet-ejb-module-jb3.jar"/>
    <pathelement location="${lib.dir}/xdoclet-jboss-module-jb3.jar"/>
    <pathelement location="${lib.dir}/xdoclet-jmx-module-jb3.jar"/>
    <pathelement location="${lib.dir}/xdoclet-web-module-jb3.jar"/>
    <pathelement location="${jboss.dist}/client/jbossall-client.jar"/>
    <pathelement location="${jboss.dist}/client/log4j.jar"/>
    <pathelement location="${jboss-net.sar}/commons-logging.jar"/>
</path>

```

本文后面论述到 ejbdoclet 任务时需要引用上述内容。至此，开发者可以创建依赖 XDoclet 的 compile-src 任务了，如列表 12-3 所示。

列表 12-3 xdoclet target 定义

```
<target name="compile-src" depends="config">
  <!-- Generate the xdoclet source -->
  <property name="xdoclet.classpath" refid="xdoclet.classpath"/>
  <echo>xdoc target</echo>
  <echo message="\${xdoclet.classpath}"/>
  <taskdef
    name="ejbdoclet"
    classname="xdoclet.modules.ejb.EjbDocletTask"
    classpathref="xdoclet.classpath"
  />

  <tstamp>
    <format property="TODAY" pattern="d-MM-yy"/>
  </tstamp>

  <ejbdoclet
    destdir="\${build.src.dir}"
    excludedtags="@version,@author"
    addedtags="@xdoclet-generated at \${TODAY}"
    ejbspec="2.0"
  >
    <fileset dir="\${src.dir}">
      <include name="org/jboss/chap12/hello/*Bean.java"/>
      <include name="org/jboss/chap12/hello/*Obj.java"/>
    </fileset>

    <packageSubstitution packages="implementation"
      substituteWith="interfaces"/>

    <remoteinterface pattern="{0}"/>

    <localinterface pattern="{0}Local"/>

    <homeinterface/>
    <localhomeinterface/>

    <deploymentdescriptor destdir="\${chapter.metainf.dir}"/>
    <jbossnet webDeploymentName="HelloBean"
      prefix="hello"
      destdir="\${chapter.metainf.dir}"
      targetNamespace="http://localhost/HelloBean"/>
  </ejbdoclet>

  <java dir="\${build.src.dir}" fork="yes" failOnError="true">
```

```

        className="org.apache.axis.wsdl.WSDL2Java">
        <!-- Map the data object to the org.jboss.chap12.hello.xml pkg -->
        <arg value="-Nhttp://localhost/
        HelloBean=org.jboss.chap12.hello.xml"/>
        <!-- Map the data object to the org.jboss.chap12.hello.xml pkg -->
        <arg value="-Nhttp://localhost:8080/jboss-net/services/
        Hello=org.jboss.chap12.hello.xml"/>
        <!-- Retrieve the wsdl from the deployed ejb -->

        <arg value="{src.resources}/hello.wsdl"/>
        <classpath refid="axis.path"/>
    </java>
</target>

```

上述列表给出了编译 EJB 和运行 XDoclet 的基本内容。接下来，本文来分析一下该列表。首先，它声明了 ejbdoclet 任务定义。请注意，这里传入前面定义的 xdoclet.classpath 到 classpathref 中。其次，声明了时间戳（tstamp）属性。本文将它直接放在 ejbdoclet 任务下面，供 XDoclet 生成文件时打印出时间信息，以表明何时创建了它们。请注意，该 target 的剩余部分封装在 ejbdoclet 任务里面。destDir 属性继承了顶级 build.xml 文件中的 build.src.dir，即指定 output/gen-src 目录。另外，还特别使用了 excludedtags 属性，将一些常见目的的 javadoc 标签排除在待处理的标签之外。同时，还添加了 addedtags 属性以生成时间戳，然后借助于 ejbspec 属性指定待兼容 EJB 2.0 规范。

ejbdoclet 任务中存在着一个嵌入“fileset”任务。该任务是 Ant 本身提供的，它用于指定该上下文中 XDoclet 能够处理的文件集合。请注意，这里只处理 org/jboss/chap12/hello/*Bean.java 和 org/jboss/chap12/hello/*Obj.java 类。只有保存在 src/main/org/jboss/chap12/hello 目录中后缀名为 Bean.java 或 Obj.java 的文件才会被 ejbdoclet 处理。开发者需要遵循上述这样一种约定，从而能够正确地控制 XDoclet 的作用范围。当然，也可以使用包，或者显式地声明所有的文件名，但还是没有这里给出的方式灵活。比如，如果不打算借助于 XDoclet 处理 BaseSession，但是从 Web 服务返回的所有对象又需要借助于 XDoclet 处理。因此，将需要序列化的对象命名为*Obj.java，将所有的企业 Bean 的 Bean 类命名为*Bean.java。

接下来，本文将阐述 packageSubstitution 子任务。EJB 开发者遵循的常见约定是，借助于包将实现类（Bean 类）和相应的接口（远程/本地/Home/本地 Home）分开。既然 XDoclet 将负责生成这些接口，所以如果开发者遵循这种约定的话，则需要了解具体做法。但如果开发者不想遵循，则可以大胆地将它丢在一边。

接着，出现了远程、本地、Home 及本地 Home 接口子任务，它们都是根据指定属性（view-type，等等）来完成具体操作的。请注意，pattern=“{0}”和“{0}Local”属性/值对。其中，{0}指文件集合。因此，如果企业 Bean 命名为“HelloBean”，则远程接口是“Hello”，本地接口是“HelloLocal”。

deploymentdescriptor 子任务负责生成部署描述符。请注意，这里将描述符指向了 build/META-INF 目录。它将生成 ejb-jar.xml 文件。

最后，xdoclet-module-jboss-net.jar 添加的子任务 jbossnet 用于生成 Web 服务描述符。

webDeploymentName 属性用于指定 Axis 部署，并映射到 web-service.xml 中 deployment 元素的 name 属性。prefix 属性指定用于自定义应用类型的 XML 命名空间的 prefix 部分，targetNameSpace 属性指定其关联的 URI。destDir 属性指向 build/META-INF 目录，web-service.xml 描述符也将放入到该位置。

更多 ejbdoclet 任务的其他信息，请参考页面：

<http://xdoclet.sourceforge.net/ant/xdoclet/modules/ejb/EjbDocletTask.html>

下一步，开发者需要添加创建 EJB jar 和 Web 服务存档的目标（target）。创建 EJB jar 文件的 Ant 任务如下：

```
<target name="ejb-jar" depends="config"
  description="creates the ejb jar file">
  <mkdir dir="${chapter.dir}"/>
  <jar destfile="${chapter.dir}/hello-ejb.jar"
    basedir="${build.classes.dir}">
    <metainf dir="${chapter.metainf.dir}">
      <include name="ejb-jar.xml"/>
    </metainf>
    <include name="org/jboss/chap12/hello/**" />
  </jar>
</target>
```

该任务使用了 output/classes 目录中的 org.jboss.chap12.hello 包和 ejbdoclet 任务生成的 ejb-jar.xml 描述符。

ejbdoclet 任务的 jbossnet 子任务创建了 web-service.xml。该文件含有用于 Apache Axis 的 Web 服务。为定义 Web 服务，开发者还需要创建 Web 服务存档（Web Service Archive, WSR）。WSR 只是带有 META-INF/web-service.xml 描述符的标准 Java 存档。创建 WSR 的 target 如下：

```
<target name="ejb-wsr" depends="ejb-jar"
  description="creates the ejb wsr file for jb.net">
  <jar destfile="${chapter.dir}/hello-ejb.wsr">
    <metainf dir="${chapter.metainf.dir}">
      <include name="web-service.xml"/>
    </metainf>
  </jar>
</target>
```

至此，开发者还需要将 EJB jar 和 WSR 存档打包成 EAR 文件。其中，EAR 除了包含这两个存档外，还有一个标准 EAR application.xml 描述符。ear target 定义如下：

```
<target name="chap12-ex1-ear" depends="ejb-wsr"
  description="creates an ear file containing the correct files for jboss.net">
  <ear destfile="${chapter.dir}/chap12-ex1.ear"
    appxml="${src.root}/application.xml">
    <fileset dir="${chapter.dir}">
      <include name="hello-ejb.jar"/>
    </fileset>
  </ear>
</target>
```

```

    <include name="hello-ejb.wsr"/>
  </fileset>
</ear>
</target>

```

其中，为支持 EJB 和 Web 服务，config 任务创建了自定义 JBoss 配置，配置文件集合为“chap12”。开发者在构建实例后，能够在 JBOSS_DIST 目录中找到 server/chap12 配置文件集合。使用 chap12 配置启动 JBoss，然后借助于如下命令行工具，以部署 EJB 和 Web 服务，最后借助于 SOAP 客户端访问它。

```

[starksm@banshee9100 examples]$ ant -Dchap=chap12 -Dex=1 run-example
Buildfile: build.xml
...
run-example:

run-example1:
    [echo] Waiting for 5 seconds for deploy...
    [java] hello.hello(hello argument)
    [java] output:Hello hello argument!
    [java] 0: Hello index 0
    [java] 1: Hello index 1
    [java] 2: Hello index 2

BUILD SUCCESSFUL
Total time: 10 seconds

```

服务控制台的输出信息如下：

```

13:34:18,812 INFO [MainDeployer] Starting deployment of package: file:/C:/tmp/JBoss/jboss-3.2.3/
server/chap12/deploy/chap12-ex1.ear
13:34:18,812 INFO [EARDeployer] Init J2EE application: file:/C:/tmp/JBoss/jboss-3.2.3/server/
chap12/deploy/chap12-ex1.ear
13:34:19,078 INFO [EjbModule] Deploying Hello
13:34:19,234 INFO [StatelessSessionInstancePool] Started jboss.j2ee:jndiName=Hello,plugin=
pool,service=EJB
13:34:19,234 INFO [StatelessSessionContainer] Started jboss.j2ee:jndiName=Hello,service=EJB
13:34:19,234 INFO [EjbModule] Started jboss.j2ee:module=hello-ejb.jar,service=EjbModule
13:34:19,234 INFO [EJBDeployer] Deployed: file:/C:/tmp/JBoss/jboss-3.2.3/server/chap12/tmp/
deploy/tmp33528chap12-ex1.ear-contents/hello-ejb.jar
13:34:19,296 INFO [EARDeployer] Started J2EE application: file:/C:/tmp/JBoss/jboss-3.2.3/server/
chap12/deploy/chap12-ex1.ear
13:34:19,296 INFO [MainDeployer] Deployed package: file:/C:/tmp/JBoss/jboss-3.2.3/server/
chap12/deploy/chap12-ex1.ear
13:35:09,312 INFO [JaasSecurityManagerService] Created securityMgr=org.jboss.security.plugins.
JaasSecurityManager@42a818
13:35:09,328 INFO [JaasSecurityManagerService] setCachePolicy, c=org.jboss.util.TimedCache
Policy@178b64b
13:35:09,328 INFO [JaasSecurityManagerService] Added other, org.jboss.security.plugins.Security

```



```
DomainContext@83020 to map
13:35:09,843 INFO [STDOUT] hello, Hello index 0
13:35:09,843 INFO [STDOUT] hello, Hello index 1
13:35:09,843 INFO [STDOUT] hello, Hello index 2
```

如果开发者没有 WSDL 描述符,则可以从 jboss-net 服务获得 WSDL 描述符。对于 Hello Web 服务而言, jboss-net 框架将 WSDL 描述符放置在 `http://localhost:8080/jboss-net/services/Hello?wsdl`。 jboss-net 框架在如下 URL 安装了 Servlet, 即 “`http://localhost:8080/jboss-net/services`”, 从而允许开发者查询已部署的 Web 服务。为查询 Web 服务的 WSDL 描述符, 开发者需要追加服务名和 “?wsdl” 后缀¹。其中, 这里的服务名是由传入到企业 Bean 的 Bean 类的 `@jboss-net:web-service` 标签的 `urn` 属性定义的。

使用 WSDL, 开发者能够借助于 Apache Axis 创建 Web 服务客户端。Axis 提供了 WSDL2Java 工具, 供生成 Java 客户端存根使用, 从而能够访问 WSDL 描述符描述的服务。下节给出具体介绍。

使用 Apache Axis 创建 Web 服务客户

WSDL2Java 工具包含在 `deploy` 目录 `jboss-net.sar` 文件绑定的 Axis jar 文件中。 `chap12/build.xml` 文件包含了借助于 WSDL2Java 和 `examples/src/resources` 目录中的 `hello.wsdl` 文件创建 Axis 客户端存根的 Ant 脚本。具体如下:

```
<java dir="${build.src.dir}" fork="yes" failOnError="true"
  className="org.apache.axis.wsdl.WSDL2Java">
  <!-- Map the data object to the org.jboss.chap12.hello.xml pkg -->
  <arg value="-Nhttp://localhost/
    HelloBean=org.jboss.chap12.hello.xml"/>
  <!-- Map the data object to the org.jboss.chap12.hello.xml pkg -->
  <arg value="-Nhttp://localhost:8080/jboss-net/services/
    Hello=org.jboss.chap12.hello.xml"/>
  <!-- Use the wsdl for the deployed ejb -->

  <arg value="${src.resources}/hello.wsdl"/>
  <classpath refid="axis.path"/>
</java>
```

当使用 jbossnet 任务生成 `web-service.xml` 描述符时, 开发者将 `targetNameSpace` 属性指定为: `http://localhost/HelloBean`, 供 Web 服务的自定义类型使用。EJB 服务接口和支持类关联到如下 URI, 即 WSDL 文件生成的 “`http://localhost:8080/jboss-net/services/Hello`”。如果用户没有指定命名空间与 Java 包的映射规则, 则将使用默认转换。上述任务中, 传递给 WSDL2Java 命令的两个 -Nxxx 值被映射到 `org.jboss.chap12.hello.xml` 包。最后一个参

¹ 请注意, 当前生成的 WSDL 描述符包括所有已部署服务的类型定义, 而不仅仅是同查询服务相关的类型。这是当前的一个 Bug, 因此在不同命名空间中部署了相同类型名时会引发问题。开发者需要编辑返回的 WSDL 描述符, 以删除多余的类型。

数是 jbossnet URL, 即通过它能够获得 Hello Web 服务的 WSDL 描述符。图 12-1 给出了使用 WSDL2Java 生成的文件。这些文件实现了 Web 服务到 Java 的映射。



图 12-1 WSDL2Java 程序生成的 Web 服务文件

其中, Hello 接口提供了 Web 服务的 Java 表示。客户使用 HelloServiceLocator 工厂能够获得 Hello 实例。其余的类供实现层处理 SOAP 传输和对象绑定使用。列表 12-4 给出了基于上述生成代码的客户实例。

列表 12-4 用于 Hello EJB/Web 服务的 Axis 客户

```
package org.jboss.chap12.client;

import org.jboss.chap12.hello.xml.HelloObj;
import org.jboss.chap12.hello.xml.HelloReplyObj;
import org.jboss.chap12.hello.xml.Hello;
import org.jboss.chap12.hello.xml.HelloServiceLocator;

/**
 * @author Scott.Stark@jboss.org
 * @version $Revision: 1.1 $
 */
public class HelloClient
{
    public static void main(String[] args) throws Exception
    {
        HelloServiceLocator locator = new HelloServiceLocator();
        Hello hello = locator.getHello();
        System.out.println("hello.hello(" + args[0] + ")");
        System.out.println("output:" + hello.hello(args[0]));
        Object[] query = new Object[3];
        HelloObj ho = new HelloObj();
        ho.setMsg("Hello index 0");
        query[0] = ho;
        ho = new HelloObj();
        ho.setMsg("Hello index 1");
```

```
query[1] = ho;
ho = new HelloObj();
ho.setMsg("Hello index 2");
query[2] = ho;
Object[] reply = hello.complexHello(query);
for(int n = 0; n < reply.length; n++)
{
    HelloReplyObj r = (HelloReplyObj) reply[n];
    System.out.println(r.getMsg());
}
}
```

附录 A JBoss Group 和我们的 LGPL 授权

关于 JBoss Group

JBoss Group LLC 是由 Marc Fleury 创立，位于亚特兰大的专业服务公司。Marc Fleury 是 JBoss，基于 J2EE 的开源 Web 应用服务器的创始者和首席开发者。JBoss Group 集合 JBoss 核心开发队伍提供服务，比如培训、支持、咨询，以及管理 JBoss 软件和服务 Affiliate Program。通过这些商业行为资助免费的主流 JBoss 服务器开发。更多 JBoss Group 信息，请参考 JBoss 网站：

<http://www.jboss.org/services/services.jsp>。

GNU LGPL

JBoss 源代码由 LGPL (Lesser General Public License, LGPL) 授权，LGPL 详情请参考：<http://www.gnu.org/copyleft/lesser.txt>。所有 `org.jboss.*` 包命名空间中的代码都受 LGPL 保护。列表 A-1 给出了 LGPL 授权的完整内容。

列表 A-1 GNU LGPL 内容

GNU LESSER GENERAL PUBLIC LICENSE

Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc.

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of non-free programs enables many more people to use the whole GNUfree software. For example, permission to use the GNU C Library in operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the later must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed

by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work formaking modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or atable of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution

displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library

together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED,

INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the library's name and a brief idea of what it does.>

Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or License as published by the Free Software Foundation; either modify it under the terms of the GNU Lesser General Public version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library 'Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990

Ty Coon, President of Vice

That's all there is to it!

附录 B JBoss DTDs

JBoss 使用的所有 DTD 文件都存放在 “<jboss-dist>/docs/dtd” 目录，从这里用户可以获得 JBoss 发布版的最新 DTD 文件。自从 JBoss 3.2 发布版开始，JBoss 还包括 “<jboss-dist>/docs/dtd/html-svg” 目录，它含有当前 JBoss 相关 DTD 文档所对应的格式化 html/svg¹ 文档。如果用户订购了 JBoss 提供的所有文档付费服务，DTD 文件的 PDF 版本也将提供给你们。

¹译者注：SVG 文件是矢量图文件格式，在 IE 中查看需要去 Adobe 网站下载 SVG 插件。

附录 C 实例安装

本书还提供了讨论过的实例源代码，通过本书光盘能够找到。

开发者使用例子之前，惟一要完成的定制工作就是设置 JBoss 服务器的安装位置。通过如下两种方式都可以实现。其一，手工编辑“examples”目录下的 build.xml 文件，即修改 jboss.dist 的属性值，如下粗体所示。

```
<project name="JBossBook 3.2.x examples" default="build-all" basedir=".">
  <!-- Allow override from local properties file -->
  <property file=".ant.properties" />
  <!-- Override with your JBoss/Web server bundle dist location -->
  <property name="jboss.dist" value="/tmp/JBoss/jboss-3.2.3"/>
  <property name="jboss.deploy.dir" value="${jboss.dist}/server/default/deploy"/>
```

其二，在“examples”目录下创建一个“.ant.properties”文件，其中包含 jboss.dist 定义。比如：

```
jboss.dist=/usr/JBoss3.2/jboss-3.2/build/output/jboss-3.2.3
```

运行实例的 JBoss 服务器版本必须和本书实例所要求的一致。如果遇到如下问题，则首先去看看是否出现如下输出：

```
validate:
[java] ImplementationTitle: JBoss-3.0.8 CVSTag=JBoss_3_0_8
[java] WARNING: requested version: 3.2 does not match the run.jar version: 3.0.8
[java] ImplementationVendor: JBoss Group, LLC
[java] ImplementationVersion: 3.0.8 (CVSTag=JBoss_3_0_8 Date=200306050849)
[java] SpecificationTitle: JBoss-3.0.8
[java] SpecificationVendor: JBoss Group, LLC
[java] SpecificationVersion: 3.0.8
[java] JBoss version is: 3.0.8
[java] Java Result: 1
```


附录 D 索引

A

- AbstractWebContainer, 380
 - 子类 (Subclassing), 386
- Apache
 - AJP连接器, 405
 - Tomcat, 405
- ApplicationDeadlockException, 230, 231
- 认证 (Authentication), 316
 - 客户端认证步骤, 329
- 认证和授权, 316
- 授权 (Authorization), 316

B

- BMP, 236
- 构建
 - JBoss发布版, 15

C

- Catalina 参考Tomcat-4.x
- 类装载 (Class Loading)
 - 部署器, 129
 - Web应用存档 (WAR), 130
- ClassCastException, 27
- 显示类信息, 43
- IllegalAccessException, 32
- Java类型系统, 26
- JBoss 3.x架构, 41
- LinkageError, 34
- 版本化 (Versioning), 44
- 查看包的类装载器, 43
- ClassLoader
 - 架构, 26
- Classpath
 - 维护Jar顺序, 74
- ClientLoginModule, 346
- 群集, (cluster) 236
- CMP, 229, 236, 423
 - 事务对性能的影响, 477
 - 容器管理关系, 446
 - 评审实体访问, 440
 - 自定义BMP finder, 464
 - 定制jbosscomp-jdbc.xml, 430
 - 自定义数据库, 491
- Declared SQL, 460
- DVC, 441
- Dynamic JBossQL, 459
- 实体Bean列映射, 438

- 自定义实体Bean, 431
- 实体Bean的read-only域, 440
- 实例代码位置, 423
- JBossCMP DTD, 431
- JBossQL, 457
- JBossQL LIMIT和OFFSET, 458
- 优化装载, 458
- 乐观锁, 480
- 覆盖EJB-QL SQL, 457
- 主键生成, 485
- 要求的DOCTYPE, 407
- Version 2.0, 485

CMP关系

- 外键, 450
- 映射, 449
- 提交选项 (commit-option), 228
- 配置
 - default, 3
- 容器管理持久化, 423
- 也请参考CMP.

CVS

- 访问JBoss代码, 14
- JBoss代码树, 17

D

- 数据库 (Database)
 - 配置连接管理器, 298
- 数据库
 - 实例配置, 297
- DatabaseServerLoginModule, 343
- 死锁 (Deadlock)
 - 检测, 230
- 死锁 (deadlock), 227
- 部署
 - 依赖性 (Dependency), 78, 110
 - 排序 (Ordering), 110
- 描述符
 - jbosscomp-jdbc.xml, 430
- 分离式Invoker
 - 定义 (definition), 132
- 脏读 (dirty read), 228
- 动态MBean, 24
 - 实例, 111
- 动态代理 (Dynamic proxy), 192

E

- EJB

客户拦截器配置, 193
 配置Invoker监听端口, 201
 配置使用RMI/HTTP, 201
 容器缓存配置, 211
 容器提交选项配置, 214
 容器配置, 208
 容器拦截器配置, 210
 容器锁策略配置, 213
 容器持久化配置, 213
 容器插件式框架, 215
 Deployer MBean, 204
 DTD验证, 204
 实例池配置, 210
 本地引用 参考ejb-local-ref
 方法许可, 312
 引用 参考ejb-ref
 验证器, 205
 ejb-jar.xml
 ENC元素, 152
 Security元素, 307
 ejb-local-ref, 159
 ejb-ref, 156
 JBoss描述符, 158
 ENC, 152
 UserTransaction, 189
 ENC 也请参考JNDI应用组件
 环境
 env-entry, 155
F
 防火墙
 JBoss默认端口, 376
H
 HTTP
 基于RMI/HTTP访问EJB, 145
 使用JNDI, 170
 HTTPS
 为客户配置使用, 174
I
 IdentityLoginModule, 336
 Instance Per Transaction, 234
 拦截器
 客户sid, 191
 Invoker
 群集RMI/JRMP, 145, 202
 RMI/HTTP, 145
 RMI/IIOP, 144
 RMI/JRMP, 143
 Invoker
 分离式, 132
J
 JAAS

认证, 317
 介绍 (Introduction to), 316
 登录代码 (Login code), 318
 LoginModule, 319
 Principal, 317
 Subject, 317
 JBoss
 从网络启动 (Booting from the network), 10
 从代码构建, 14
 客户jar, 3
 访问CVS, 14
 发布版结构, 14
 生效安全性声明, 316
 安装二进制, 2
 授权 (license), 507
 运行选项 (run option), 10
 保护对服务器的访问, 377
 安全性模型 (security model), 321
 源代码树结构 (Source tree structure), 17
 测试套件, 17
 default配置, 3
 JBoss Group
 关于, 507
 JBoss消息
 客户jar, 239
 默认目的地, 239
 jbosscmp-jdbc.xml
 DTD, 431
 结构 (Structure), 430
 JBossCX
 架构, 287
 jbossmq-destinations-service.xml, 264
 jbossmq-service.xml, 264
 jbossmq-state.xml, 264
 JBossNS
 架构 (Architecture), 164
 JBossQL
 函数 (Function), 457
 扩展EJB QL, 457
 JBossSecurity, 328
 架构, 328
 jboss-service.xml
 DTD, 73
 JBossSX
 自定义安全性代理, 325
 登录模块, 336
 MBean, 331
 Subject使用模式, 348
 JBossTX
 适配事务管理器, 188
 内核 (Internal), 188
 jboss-web.xml
 context-root, 379

DTD图 (DTD Graphic), 379
 ENC元素, 152
 virtual-host, 379
 jboss.xml, 191, 205, 208
 客户拦截器内容模型 (client interceptor schema), 193
 client-interceptors, 192
 commit-option, 214
 容器配置, 208
 container-configuration元素, 229
 container-interceptors, 210
 container-invoker, 210
 ENC元素, 152
 instance-cache, 211
 instance-pool, 210
 locking-policy, 213
 行锁 (row-locking), 236
 security-domain, 214
 JCA, 285
 公共客户接口 (Common Client Interface), 285
 概述, 285
 实例适配器, 291
 JDBC
 配置连接管理器, 289
 实例数据源配置, 289
 实例驱动配置 (sample driver configuration), 306
 用于认证或授权, 340
 JMS
 配置queue, 276
 配置topic, 278
 连接工厂名, 239
 目的地管理, 276
 目的地管理器统计, 277
 实例, 239
 将消息持久化到数据库中, 275
 推荐的调用层, 267
 queue统计, 276
 topic统计, 278
 使用SSL, 268
 jms-ds.xml, 264
 JMX
 作为微内核, 71
 命令行访问, 66
 使用RMI连接, 56
 使用任何协议连接, 71
 MBean, 24
 保护控制台应用, 55
 SNMP事件, 131
 扩展JBoss服务, 72
 Web控制台Applet, 7
 Web控制台应用, 53
 JNDI
 应用组件环境, 151

群集环境中的查找, 168
 ENC约定, 152
 ENC 参考JNDI应用组件环境
 ExternalContext MBean, 179
 InitialContext工厂, 166
 JBoss jndi.properties设置, 166, 168
 登录 (Logging in with), 169
 MBean, 179
 NamingAlias MBean, 181
 NamingService MBean, 164
 基于HTTP, 170
 概述 (Overview), 149
 保护, 176
 保护和只读, 178
 浏览MBean, 182
 JSSE
 jar, 371
 JBoss和SSL, 367
 JTA
 默认MBean, 189
 UserTransaction, 189
 XidFactory MBean, 189
 J2EE
 安全性声明概述, 307
K
 keystore, 371
L
 LDAP
 微软活动目录 (MS Active Directory), 340
 用于认证或授权, 340
 LdapLoginModule, 339
 LGPL, 507
 日志功能 (Logging)
 改变实现 (Changing the implementation), 421
 框架类, 420
 Log4j, 420
 Log4j配置, 421
 Stderr, stdout, 421
 登录模块
 介绍 (Introduction), 336
 登录模块
 开发自定义登录模块, 347
 登录模块 参考JAAS
M
 MBean
 属性持久化, 49
 使用EJB, 111
 属性和PropertyEditor, 74
 JBoss服务, 72
 JBoss服务 参考MBean服务
 org.jboss.deployment.SARDeployer, 73

- org.jboss.ejb.EJBDeployer, 204
- org.jboss.invocation.http.server.HttpInvoker, 145
- org.jboss.invocation.jrmp.server.JRMPInvoker, 143
- org.jboss.invocation.jrmp.server.JRMPInvokerHA, 145
- org.jboss.invocation.pooled.server.PooledInvoker, 143
- org.jboss.invocation.server.HttpProxyFactory, 146
- org.jboss.jms.asf.ServerSessionPoolLoader, 282
- org.jboss.jms.jndi.JMSProviderLoader, 281
- org.jboss.jmx.adaptor.snmp.agent.SnmpAgentService, 131
- org.jboss.jmx.adaptor.snmp.trapd.TrapdService, 132
- org.jboss.logging.Log4jService, 421
- org.jboss.mq.il.jvm.JVMServerILService, 265
- org.jboss.mq.il.oil.OILServerILService, 266
- org.jboss.mq.il.rmi.RMIServerILService, 265
- org.jboss.mq.il.uil.UILServerILService, 266
- org.jboss.mq.pm.file.CacheStore, 274
- org.jboss.mq.pm.file.PersistenceManager, 274
- org.jboss.mq.pm.jdbc2.PersistenceManager, 275
- org.jboss.mq.pm.rollinglogged.PersistenceManager, 274
- org.jboss.mq.security.SecurityManager, 271
- org.jboss.mq.server.jmx.DestinationManager, 263
- org.jboss.mq.server.jmx.InterceptorLoader, 270
- org.jboss.mq.server.jmx.Invoker, 269
- org.jboss.mq.server.jmx.Queue, 276
- org.jboss.mq.server.jmx.Topic, 278
- org.jboss.mq.server.MessageCache, 274
- org.jboss.mq.sm.file.DynamicStateManager, 270
- org.jboss.naming.ExternalContext, 179
- org.jboss.naming.JNDIView, 182
- org.jboss.naming.NamingAlias, 181
- org.jboss.naming.NamingService, 164
- org.jboss.resource.connectionmanager.BaseConnectionManager2, 288
- org.jboss.resource.connectionmanager.CachedConnectionManager, 291
- org.jboss.resource.connectionmanager.JBossManagedConnectionPool, 290
- org.jboss.resource.connectionmanager.LocalTxConnectionManager, 299
- org.jboss.resource.connectionmanager.NoTxConnectionManager, 299
- org.jboss.resource.connectionmanager.RARDeployment, 289
- org.jboss.resource.connectionmanager.XATxConnectionManager, 299
- org.jboss.resource.RARDeployer, 287
- org.jboss.security.auth.login.XMLLoginConfig, 334
- org.jboss.security.plugins.JaasSecurityDomain, 333
- org.jboss.security.plugins.JaasSecurityManagerService, 331
- org.jboss.security.plugins.SecurityConfig, 336
- org.jboss.security.srp.SRPVerifierStoreService, 360
- org.jboss.security.srp.SRPService, 366
- org.jboss.services.binding.ServiceBindingManager, 410
- org.jboss.system.ServiceController, 77
- org.jboss.tm.TransactionManagerService, 189
- org.jboss.tm.usertx.server.ClientUserTransactionService, 189
- org.jboss.tm.XidFactory, 189
- org.jboss.varia.property.SystemPropertiesService, 409
- org.jboss.varia.scheduler.Scheduler, 417
- org.jboss.web.WebService, 521
- 在属性值中引用系统属性, 74
- 指定依赖性 (Specifying dependency), 78
- MBean服务, 72
 - 部署描述符DTD, 73
 - 服务生命周期 (service lifecycle), 76
- MBean
 - 检查依赖性状态 (Inspecting dependency status), 78
 - org.jboss.proxy.generic.ProxyFactoryHA, 145
 - 标准MBean实例, 82
- 方法许可, 312
- 模型MBean, 24
 - JBoss XMBean实现, 47
- N**
 - 命名
 - 参考JNDI
 - 网络启动 (Netboot), 10
- O**
 - 开放MBean (Open MBean), 24
- P**
 - 挂起 (Passivation)
 - 设置超时 (timeout setting), 212
 - 属性 (Properties)
 - 管理 (Managing), 409
 - 系统 (System), 409
 - ProxyLoginModule, 345
- R**
 - read-only, 233
 - 可重复读 (repeatable read), 233
 - 资源适配器 参考JBossCX
 - resource-env-ref, 163
 - JBoss描述符, 164
 - resource-ref, 161
 - JBoss描述符, 162
 - RFC2945 参考SRP
 - RMI
 - HTTP实例配置, 201
 - JRMP压缩Socket实例 (JRMP compressed socket example), 199
 - 基于SSL, 369
 - 回滚 (rollback), 228, 230
 - RunAsLoginModule, 346

S

SAR

定义 (definition), 73

定时 (Scheduling), 417

安全性

JDBC, 343

失效缓存 (Disabling caching), 333

EJB许可 (EJB permission), 312

用于EJB和WAR生效, 323

用于JBoss生效, 316

扩展登录配置, 347

刷新认证信息, 333

介绍JAAS, 316

JBoss架构, 328

J2EE, 307

列举安全性域中的活动用户, 333

设置缓存策略, 332

客户认证的步骤, 329

JBoss模型, 321

使用LDAP, 339

Web内容许可, 315

安全性管理器

运行 (Running with), 367

security-constraint, 315

security-identity, 310

security-role, 311

security-role-ref, 309

Servlet容器

集成, 379

SNMP

展示事件, 131

SRP, 357

算法, 362

实例, 364

集成安全性数据, 360

JBossSX特征, 357

JBossSX实现, 357

登录模块 (login module), 336

实例登录配置, 358

SRPLoginModule选项, 358

SRPLoginModule, 358

SSL

EJB, 369

JaasSecurityDomain, 333

Tomcat-4.x, 388

JSSE, 369

标准MBean (Standard MBean), 24

standardjbosscmp-jdbc.xml, 430

standardjboss.xml, 191, 205, 207, 208

启动

过程, 71

启动类

参考JBoss服务

T

定时器, 417

Tomcat

服务描述符, 389

Tomcat-4.x, 388

Apache, 405

虚拟主机, 379

群集, 407

配置, 389

设置SSL, 394

Transaction

概述, 185

事务, 228 229

U

UCL. 参考UnifiedClassLoader3

UnifiedClassLoader3, 42

UnifiedLoaderRepository3, 42

UsersRolesLoginModule, 338

UserTransaction

支持 (Support), 189

V

virtual-host 参考 jboss-web.xml

W

WAR

设置上下文路径, 379

web.xml

ENC元素 153

Security元素, 307

X

XMBean

JBoss ModelMBean实现, 47

jboss_xmbean_1_0 DTD, 47